Proposta de Reengenharia para um Sistema Web POS/ERP

Luiz Henrique Freire Barros

CENTRO DE INFORMÁTICA
UNIVERSIDADE FEDERAL DA PARAÍBA

JOÃO PESSOA - 2019

Luiz Henrique Freire Barros

Proposta de Reengenharia para um Sistema Web POS/ERP

Monografia apresentada ao curso Ciência da Computação do Centro de Informática, da Universidade Federal da Paraíba, como requisito para a obtenção do grau de Bacharel em Ciência da Computação.

Orientadora: Profª. Dra. Danielle Rousy Dias da Silva

Maio de 2019

Catalogação na publicação
Seção de Catalogação e Classificação

B277p Barros, Luiz Henrique Freire.
Proposta de Reengenharia para um Sistema Web POS/ERP / Luiz Henrique Freire Barros. - João Pessoa, 2019.
93 f. : il.

Orientação: Danielly Rousy Dias da Silva.
Monografia (Graduação) - UFPB/CI.

1. Sistemas Legados. 2. Sistemas POS. 3. Modelos Arquiteturais. 4. Modelo de Banco de Dados. 5. Frameworks Web. I. da Silva, Danielly Rousy Dias. II. Título.

UFPB/CI

CENTRO DE INFORMÁTICA

UNIVERSIDADE FEDERAL DA PARAÍBA

Trabalho de Conclusão de Curso de Ciência da Computação intitulado Proposta de Reengenharia para um Sistema Web POS/ERP de autoria de Luiz Henrique Freire Barros, aprovada pela banca examinadora constituída pelos seguintes professores:

Danielle Rousy Dias da Silva

Profª. Dr. Danielle Rousy Dias da Silva

CI/UFPB

Eudisley Gomes dos Anjos

Prof Dr. Eudisley Gomes dos Anjos

CI/UFPB

Mardson Freitas de A

Prof. Dr. Mardson Freitas de Amorim

CI/UFPB

João Pessoa, 13 de maio de 2019.

Centro de Informática, Universidade Federal da Paraíba

Rua dos Escoteiros, Mangabeira VII, João Pessoa, Paraíba, Brasil CEP: 58058-600

Fone: +55 (83) 3216 7093 / Fax: +55 (83) 3216 7117

"Não é o crítico que conta; Não o homem que aponta como o homem forte tropeça, ou onde o fazedor de ações poderia ter feito melhor. O crédito pertence ao homem que está realmente na arena, cuja face está manchada pela poeira e suor e sangue; Que se esforça valentemente; Que erra, que 'quase chega lá' repetidamente, porque não há nenhum esforço sem erro ou falha;

Mas quem realmente se esforça para fazer as obras; Que conhece grande entusiasmo, e grande devoção; Que se consome numa causa digna; Que, no melhor dos casos, conhece no final o triunfo da alta realização e que, no pior dos casos, se falhar, pelo menos falhará tendo ousado muito, de modo a que o seu lugar nunca estará com aquelas almas frias e tímidas que não conhecem a vitória ou a derrota."

Theodore Roosevelt - O homem na Arena

AGRADECIMENTOS

Agradeço aos meus pais Roberta Maria de Miranda Freire e Augusto do Rêgo Barros Filho por me proporcionar uma educação de primeira qualidade mesmo em situações difíceis e me apoiarem durante toda a minha trajetória e decisões.

Aos meus colegas de curso, em especial a Aline Moura Araújo, Alisson Galiza Pires Martins, Emanuel Albuquerque, Gabriel da Silva Berlamino e Rhenan Castelo Branco por enfrentarem comigo as dificuldades da vida acadêmica, pelos ensinamentos e pelos momentos descontraídos.

Aos meus professores do Centro de Informática pela formação e inspiração que me proporcionaram. Em especial ao professor Mardson Freitas de Amorim, por me apresentar a área de computação e ao professor Tiago Maritan Ugulino de Araújo, por me proporcionar meus primeiros contatos com o ambiente de trabalho, tanto no LAViD como na Assista Tecnologia.

RESUMO

Com o passar do tempo, o código de um software tende a perder a qualidade por diversos motivos, principalmente em códigos legados (aqueles que já estão em uso há bastante tempo e que usam tecnologia supostamente ultrapassada). E, às vezes, se faz necessário uma reengenharia no software a fim melhorar alguma propriedade, seja o desempenho, a escalabilidade ou testabilidade. Este trabalho propõe uma reengenharia de um sistema legado analisando pontos como framework e bibliotecas, modelo de banco de dados e modelo arquitetural. E, como resultado, foi constatado por meio de uma revisão da literatura, que, para o sistema de estudo, a melhor biblioteca para a construção da camada de interface é o React.js e o melhor modelo de banco de dados é o relacional. Sobre o modelo arquitetural, não foi possível decidir, uma vez que a sua escolha depende de outros fatores, como, por exemplo, o framework escolhido.

Palavras-chave: Sistema Legados, Sistema POS, Modelos Arquiteturais, Modelo de Banco de Dados, Frameworks Web

ABSTRACT

Over time, software code tends to lose quality for many reasons, especially in legacy codes (those that have been in use for a long time and that use supposedly outdated technology). And sometimes it takes a re-engineering in software in order to improve some property, like performance, scalability or testability. This work proposes a reengineering of a legacy system analyzing points such as frameworks and libraries, database model and architectural model. As a result, it was found through a literature review, that for the study system, the best library for the construction of the interface layer is React.js and the best database model is the relational model. About the architectural model, it was not possible to decide, since its choice depends on other factors, such as the chosen framework for example.

Key-Words: Legacy System, POS System, Architectural Models, Database Model, Web Frameworks

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS

Back-End	-	Desenvolvimento voltado à lógica
ERP	-	do inglês, *Enterprise resource planning*
Front-End	-	Desenvolvimento voltado à interface e experiência de usuário
POS	-	do inglês, *Point of Sales*
XML	-	do inglês, *Extensible Markup Language*
MVC	-	Model-View-Controller
MVP	-	Model-View-Presenter
MVVM	-	Model-View-Viewmodel
KO	-	Knockout.js
API	-	*Application Programming Interface*
HTML	-	*HyperText Markup Language*

Sumário

1. Introdução

As bases conceituais da rede mundial de computadores começaram a ser implementadas nos primeiros anos da década de 90. Desde então, sua importância cresceu de forma que o seu idealizador, Tim Berners-Lee, nunca havia imaginado (BERNERS-LEE, 1996). As aplicações estão realizando cada vez mais tarefas dentro do próprio navegador, o que implica em programas cada vez mais complexos [1]. Aliado a isso, o mercado de aplicações WEB sempre foi bastante eufórico e excitante. Novos padrões e frameworks são apresentados constantemente e, não raro, a comunidade de desenvolvedores acaba se perdendo em meio a tantas opções no mercado.

Porém, às vezes é preciso fazer uma refatoração ou até mesmo uma reegenharia em um código já existente e ainda em uso (os chamados, códigos legados) e avaliar as decisões tomadas anteriormente. Chinfosky [2] define reengenharia como o processo de análise e alteração de um sistema, reconstruindo-o e reimplementando-o com novas tecnologias, ou, em outras palavras, é a renovação ou recuperação de software. Uma reengenharia não apenas recupera informações do projeto de um sistema existente, como também usa essas informações para reconstruir o sistema, procurando melhorar sua qualidade global e reduzir custos com manutenção [3].

O objetivo da reengenharia de software é facilitar modificações subsequentes e manter o código mais limpo, otimizado e organizado. Para isso, existem 3 tipos de reengenharia, como citado em [3]:

- **Engenharia avante:** implementação tradicional de desenvolvimento de software que parte de um nível mais alto de abstração, passando pela análise de requisitos e projeto até a "programação" propriamente dita.
- **Engenharia de transformação:** Nada mais é do que o processo de migração de sistemas para ambientes mais modernos, porém mantendo a mesma funcionalidade
- **Engenharia reversa:** é uma operação contrária à engenharia avante, que tem início no código fonte de implementação do sistema legado, recuperando ou

recriando seu projeto e compreendendo os requisitos que foram implementados. Essa foi a abordagem usada neste trabalho.

Processos de reengenharia se fazem necessários em sistemas legados por diversos motivos, que, dentre alguns pode-se citar:

- **Falta de documentação:** normalmente sistemas legados não possuem nenhum tipo de documentação. Quando possuem, não é raro que eles estejam desatualizados, pois o código e os requisitos podem ter mudado com o tempo, e essas mudanças não foram documentadas, o que dificulta muito o trabalho de manutenção [3] [5].
- **Código mal feito:** por muitas vezes é melhor refazer uma seção do código a partir do zero do que tentar entender e reutilizar um código mal feito de outra pessoa [5].
- **Surgimento de tecnologias melhores:** é possível que novas tecnologias tenham surgido desde a primeira elicitação de requisitos até o atual momento, e que essas novas tecnologias atendam melhor às necessidades [5].

Atualmente, há diversos softwares legados em operação. O sistema da empresa utilizada como caso de uso nesta monografia, e que por questões éticas será tratada como Empresa X, é um desses sistemas. A empresa trabalha no ramo de roupas de luxo sob medida e tem pretensão de expandir seus negócios para outros países, tornando crucial o uso de um sistema de informação robusto. Esse ramo de negócio possui diversos processos, alguns mais genéricos e outros bastante específicos, possuindo processos de venda, acompanhamento, edição do produto, notificação de usuários, processos de prova, processos de alteração e de logística.

O sistema é dividido em duas partes, um correspondente ao lado do cliente, chamado POS (do inglês, Point of Sale) e do lado do servidor, o ERP (do inglês, Enterprise Resource Planning). Essa divisão é muito comum em empresas de varejo onde o intuito é a venda de algum bem ou serviço. Este trabalho tem como principal foco o sistema POS (lado do cliente). Hoje, esse sistema usa um padrão arquitetural MVVM, banco de dados relacional, e a principal biblioteca é o KnockoutJS. Esse sistema POS existe desde 2014, contudo, após contínuas manutenções, foi possível perceber melhorias que poderiam ser desenvolvidas para facilitar sua manutenção a longo prazo.

Baseado neste contexto, este trabalho se propõe a analisar oportunidades de reengenharia de sistema que possibilitam a facilidade de manutenção, escalabilidade, testabilidade, modularidade e flexibilidade.

1.1 Objetivos gerais

A área de desenvolvimento Web possui muitas ferramentas e fatores que influenciam a vida dos desenvolvedores. As diferentes ferramentas e conhecimentos necessários influenciam diretamente no desempenho e satisfação do usuário em relação à aplicação bem como o ritmo do desenvolvimento.

Este trabalho tem como objetivos gerais analisar um sistema legado por meio de engenharia de requisitos e avaliar as decisões tomadas na escolha de determinadas ferramentas tais como, modelo de banco de dados, frameworks e arquitetura, comparando com as tecnologias presente no mercado.

1.2 Objetivos específicos

Como objetivos específicos, pretende-se:

1. Entender o sistema POS/ERP através da análise de código
2. Extrair os requisitos do POS/ERP e especificá-los
3. Analisar o projeto do sistema com base na arquitetura, tecnologias e contextos de uso
4. Realizar um estudo comparativo entre diferentes tecnologias de desenvolvimento, tais como banco de dados, frameworks e arquiteturas;
5. Propor melhorias na arquitetura da camada de visão do sistema por meio de outro framework que utilize o modelo de sistemas SPA;

2. Fundamentação Teórica

É sabido que ao longo do tempo, o código tende a perder a qualidade. Ao poucos ele começa a apresentar código repetido, classes localizadas em pacotes errados, métodos localizados em classes erradas, entre outros erros que podem prejudicar a manutenção da estrutura interna do software. E, ao contrário do que muitos pensam, às vezes se faz necessário realizar um pouco de retrabalho e reavaliar as decisões tomadas na idealização de uma aplicação WEB [5].

Esse processo pode ser simples ou complexo dependendo do que deve ser refatorado, em geral requisitos funcionais (aqueles que ditam o que o software deve fazer) são mais simples de serem alterados. Já os requisitos não-funcionais (aqueles que ditam como o software funciona, bem como suas tecnologias) geralmente são mais difíceis de alterar e afetam bem mais o sistema [6].

As tecnologias e ferramentas escolhidas para construir a aplicação fazem parte dos requisitos não-funcionais e modificá-las pode ser bastante trabalhoso porém necessário. Para cada setor de um software, o programador de sistemas tem a sua disposição diversas ferramentas de desenvolvimento. Há diversas arquiteturas bem difundidas para elaboração de sistemas WEB, como por exemplo o MVC, MVVM e o MVP. Se ele precisa desenvolver um banco de dados, por exemplo, existem pelo menos quatro tipos de modelos diferentes, como por exemplo os bancos orientados a objeto, orientados a documento, relacional etc. Se quer fazer um sistema WEB em SPA, existem diversos frameworks e bibliotecas que trabalham com esse tipo de abordagem, por exemplo o Knockout e o ReactJs. E escolher a melhor ferramenta, nem sempre é uma tarefa tão trivial.

Nesta seção, será discutido um pouco mais sobre sistemas legados e reengenharia de software, além de apresentar algumas tecnologias de desenvolvimento WEB. Será discutido também, arquiteturas VM*, frameworks para abordagem SPA e modelo de banco de dados, bem como um comparativo entre as tecnologias, mostrando em que situações cada uma é mais viável do que a outra.

2.1 Sistemas Legados

Por definição, sistemas legados são sistemas críticos em uso há determinado período por uma empresa e desenvolvidos com tecnologias supostamente ultrapassadas [7].

Não é difícil identificar um sistema legado. Segundo PINTO (2014), existem algumas características inerentes a esse tipo de sistemas. São elas:

- Sistemas em produção há mais de 5 anos;
- Hardware e/ou software obsoletos;
- Sistemas com mais de 10 mil linhas de códigos;
- Documentação antiga ou desatualizada (ou ainda inexistente);
- Código modificado por diversas equipes ao longo do tempo;
- Gerenciador de banco de dados obsoleto;
- Regras não estão documentadas ou não são conhecidas pela grande maioria da equipe de manutenção;

No que se refere à evolução de um sistema legado, ele pode ser dividido em 3 etapas: manutenção, substituição e modernização. A **Figura 01** abaixo ilustra o ciclo de vida de uma aplicação e as diferentes fases.

Mesmo depois que o sistema é entregue, os requisitos podem continuar mudando. É na fase da manutenção, em que o software é incrementado, e pequenas modificações são inseridas. Essas modificações podem ser erros ou pequenas melhorias no sistema, mas não indicam grandes mudanças estruturais [7].

A modernização já implica em melhorias mais significativas, como a troca de alguns poucos requisitos não-funcionais, porém grande parte do sistema é mantido [7].

Por fim, quando a modernização não é mais possível, inicia-se o processo de substituição, ou seja, uma troca completa do sistema, uma vez que ele já não consegue mais atender às necessidades do negócio [7].

O sistema da empresa X se encaixa em quase todas as características de um sistema legado citadas acima, e o que esta monografia sugere é que se inicie a **fase de substituição** do software. Para isso, algumas partes do projeto, senão todo ele, precisam ser avaliadas, desde a

escolha de bibliotecas até a linguagem de programação usada no software. Nesta monografia, o foco está em apenas três tecnologias: framework do *front-end*; modelo de banco de dados; e arquitetura de software.

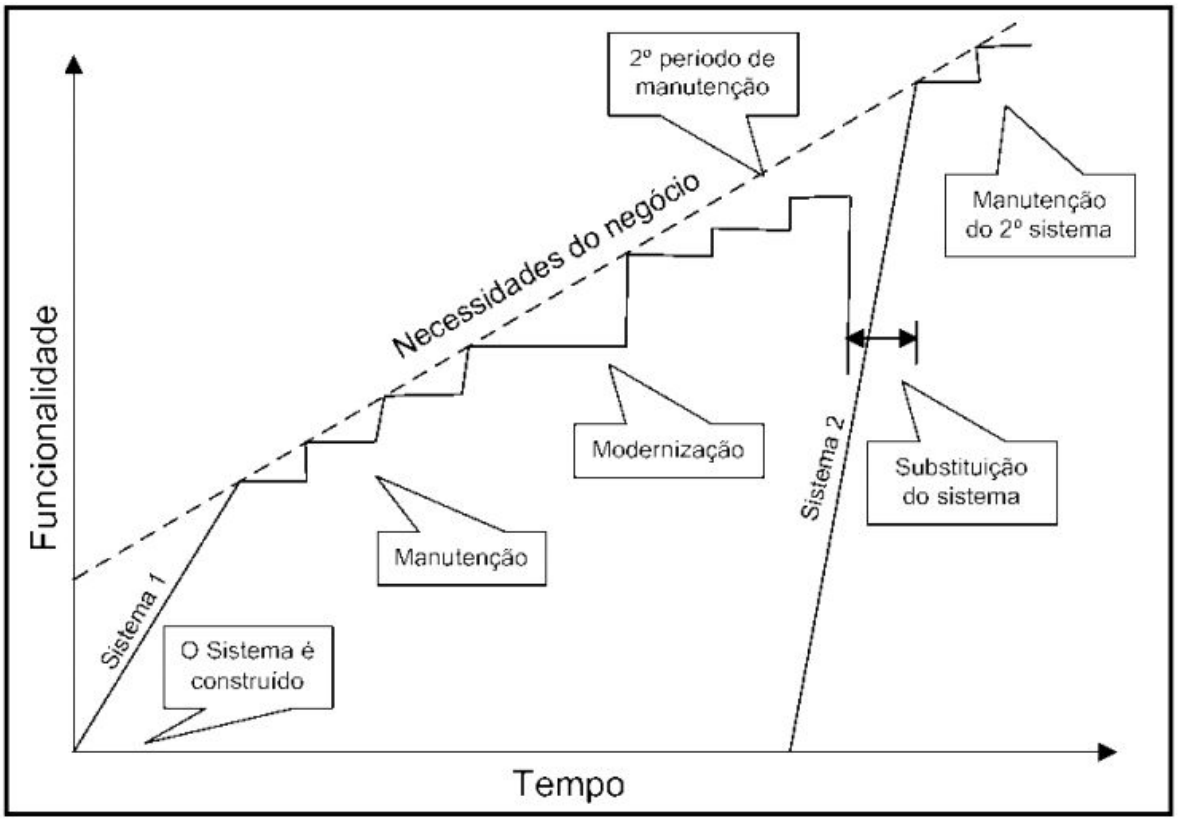


Figura 01 - Ciclo de vida de um sistema legado [7]

2.2 Arquiteturas de Software

Uma arquitetura de software consiste na definição dos componentes da aplicação, suas propriedades externas, e seus relacionamentos com outros softwares. Como formalmente definido pela (ISO/IEEE 1471-2000), "Arquitetura é a organização fundamental de um sistema incorporada em seus componentes, seus relacionamentos com o ambiente, e os princípios que conduzem seu design e evolução". Seja qual for o produto, uma das primeiras decisões de um desenvolvimento é definir sua arquitetura, sendo também muito importante numa etapa de reengenharia. Essa atividade é muito importante para garantir diversas

qualidades no código, tais como divisão de responsabilidades entre membros da equipe, tolerância a falhas, escalabilidade, reuso, tolerância a mudanças, entre outros.

De forma geral, pode-se dizer que as arquiteturas que se assemelham à usada pela empresa X são divididas em três camadas:

- **Camada de visão ou de apresentação, ou *view***: Responsável pelos elementos gráficos que interagem com o usuário, tais como botões, formulários, animações etc.
- **Camada lógica de negócio**: Responsável pela implementação da lógica de negócio. É o "coração" da aplicação, ou seja, onde a maior parte do processamento é feito.
- **Camada de modelo ou de dados, ou *model***: Esta camada recebe as requisições da camada de negócios e seus métodos executam essas requisições em um banco de dados. Uma alteração no banco de dados alteraria apenas as classes da camada de dados, mas o restante da arquitetura não seria afetado por essa alteração.

O que as arquiteturas diferem uma da outra, normalmente, é como esses componentes se conectam e cada uma delas serve para um propósito.

2.2.1 Arquitetura MVC Original

O MVC vem do inglês *model-view-controller*, que sugere que a aplicação tenha essas três camadas. O modelo é a camada relacionada com o lógica de negócio. A visão é a camada que exibe os elementos do modelo na tela, ou seja, é por ela que o usuário interage com o sistema. Por fim, os controladores agem como um intermediário entre as outras camadas [8]. Veja a **Figura 02**.

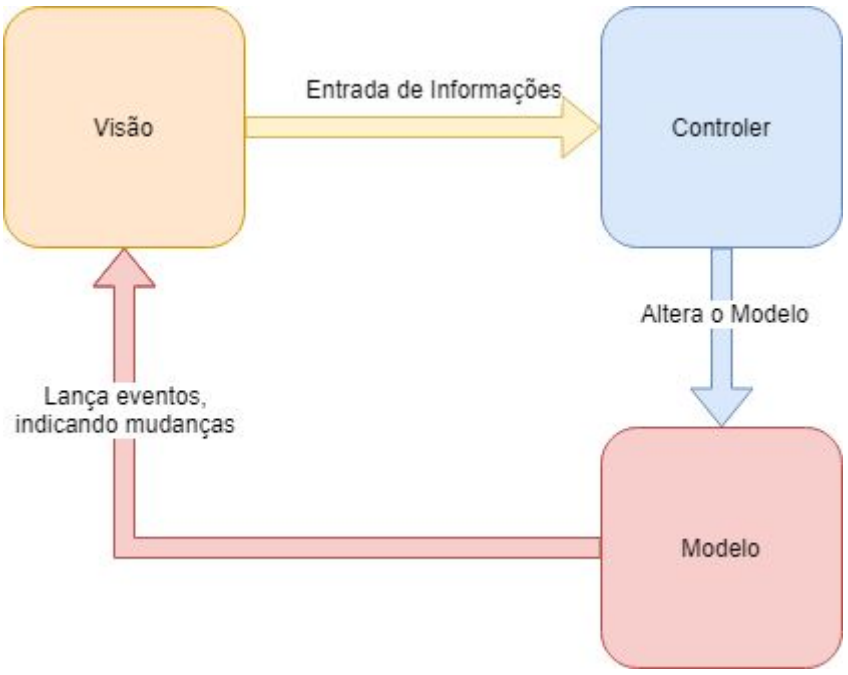


Figura 02 - Representação gráfica do MVC
adaptado de [9]

O principal ganho desse tipo de arquitetura é o reuso, a possibilidade de representar o mesmo modelo de diferentes formas. Porém, uma das desvantagens é que, apesar das camadas serem bem definidas, esse tipo de modelo gera uma certa dependência entre a camada do modelo e a visão [8].

Com isso, surgem alguns problemas com esse padrão como:

- Modularidade e flexibilidade - como os controladores estão acoplados às views, se a view muda, deve-se voltar e mudar o controlador também [10].
- Manutenção - Ao longo do tempo, em detrimento a sucessivas modificações e manutenções, cada vez mais o código começa a ser transferido para os controladores, tornando-os cheios, complexos e aumentando as chances de falhas [10].

Com o tempo, os sistemas da informação foram mudando. A complexidade dos sistemas foi crescendo à medida que as interfaces gráficas foram ficando mais elaboradas [11]. Muitos padrões surgiram derivados do MVC. Um que focou na separação entre as

classes de modelo e as classes de visão foi o MVVM que será abordado na seção 2.2.3 mais à frente.

2.2.2 Arquitetura MVP

Enquanto alguns autores defendem que o MVP é um novo padrão, existem outros que defendem que ele é apenas uma nova forma de interpretar o MVC [12] devido à grande semelhança entre esses dois padrões arquiteturais.

Assim como no MVC, o MVP tem 3 camadas distintas:

- O Model, que é exatamente igual ao do MVC
- A View, assim como no MVC, também representa os componentes de interface, ou seja, aqueles elementos dos quais o usuário interage diretamente. Porém, diferentemente do padrão MVC, esta camada é totalmente separada do da camada de modelo. Ver **Figura 03**.
- Por fim, o Presenter, que é basicamente o controlador do MVC, exceto por ele não estar vinculado à visão, e sim a uma interface entre ele e a View. Assim, a dependência com a camada que se comunica com o usuário é amenizada, além das preocupações de testabilidade, modularidade e flexibilidade serem melhores atendidas quando comparado com o padrão MVC [12].

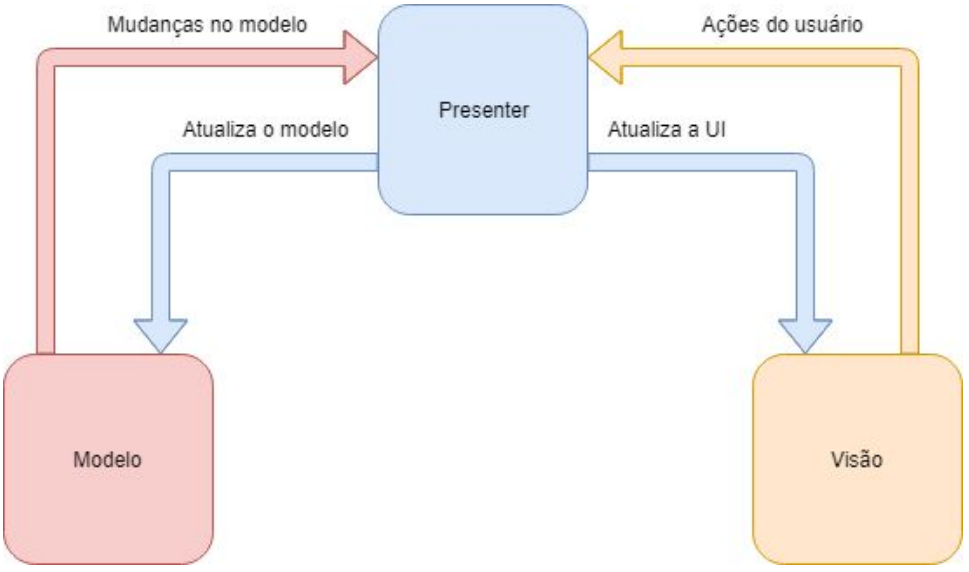


Figura 03 - Representação gráfica do MVP [10]

2.2.3 Arquitetura MVVM

A arquitetura MVVM foi divulgada inicialmente em 2005, no blog de John Gossman, na época arquiteto do Windows Presentation Foundation (WPF) e Silverlight na Microsoft. Neste padrão, assim como no MVC, o modelo contém os dados e a lógica de negócio e a visão é responsável pela interface gráfica. Porém, neste tipo de arquitetura, para cada visão, existe uma camada intermediária entre a *view* e o modelo, o *viewModel*. Essas duas camadas se comunicam entre si através de um mecanismo vinculação de dados, de tal forma que, se o modelo da visão mudar, ela será notificada e atualizada, e vice-versa. Essa vinculação (chamada de *data-bindI),* normalmente é implementada por meio do padrão Observer[1] [8]. Idealmente, essa vinculação é feita através de frameworks ou bibliotecas. Ver **Figura 04**.

[1] Um padrão de projeto utilizado quando se pretende que um ou mais componentes faça algo quando outro se modifica [44]

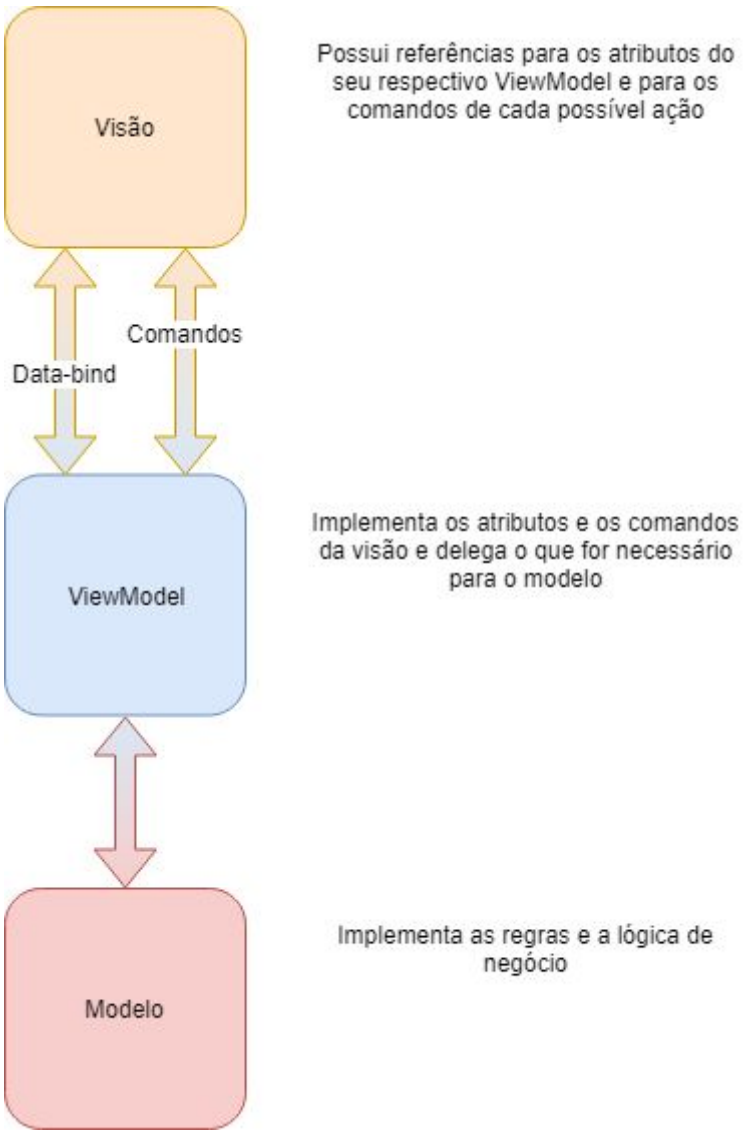


Figura 04 - Representação gráfica do MVVM [8]

Desta forma, a viewModel atua como uma peça fundamental para esse modelo, uma vez que serve como ponte entre a visão e o modelo (elementos que não se comunicam diretamente).

2.3 Aplicação WEB

2.3.1 JavaScript

Embora o motivo inicial tenha sido meramente o registro e disseminação de artigos acadêmicos, a World Wide Web, ou "rede mundial de computadores", mudou desde o seu nascimento na década de 90 [13]. À medida que ela foi se popularizando, a internet tomou um importante papel tanto econômico, como político e social e, com isso, a complexidade das aplicações também aumentou consideravelmente. Graças a tecnologias como o JavaScript e ao Ajax, páginas que antes se resumiam a documentos estáticos, hoje são comparadas a aplicações nativas em termos de desempenho [14].

O JavaScript foi criado em 1995, por Brendan Eich devido à necessidade de permitir uma maior interatividade em páginas web. Inicialmente, ele apenas era responsável pela validação de entrada em formulários (identificar se o campo a ser preenchido está correto, ou em branco, por exemplo), uma vez que antes dele, era necessário uma chamada para o servidor para realizar tal tarefa. Atualmente, o JavaScript já não está vinculado somente para validação de dados simples, agora interage com quase todos os aspectos da janela do navegador e seus conteúdos. O JavaScript é reconhecido como uma linguagem de programação completa, capaz de cálculos e interações bastante complexas [14].

Porém, o desenvolvimento dessa linguagem não foi tão simples assim. Os navegadores possuíam a sua própria linguagem de *scripting,* e à medida que os navegadores foram se desenvolvendo, surgiram também, discordâncias entre um navegador e outro. Essas discordâncias motivaram a comunidade a desenvolver APIs que unificassem o código, fazendo com que eles funcionassem em diversos tipos de *browsers* [1]. Foi assim que surgiram bibliotecas com foco em prover funcionalidade uniforme através de diversos navegadores, como jQuery [15].

O problema da divergência entre os *browsers* começou a ser solucionado em 2008, onde os representantes dos maiores fornecedores de navegadores web se reuniram com o objetivo de criar um padrão. A esse projeto foi dado o nome de ECMAScript Harmony [1] [16]

2.3.2 HTML, CSS e DOM

HTML e CSS são ferramentas básicas para qualquer página WEB. Segundo CECHINEL (2017), "O HTML é uma linguagem de marcação de hipertexto. Hipertextos são conjuntos de elementos interligados através de hyperlinks[2], podendo esses elementos ser palavras, imagens, vídeos, áudio, documentos etc".

Já o CSS, também definido em [17], "é uma linguagem de estilo utilizada para descrever a aparência e a formatação de um documento escrito em uma linguagem de marcação" (no caso, o HTML). Foi proposta por Hakon Wium Lie, em 10 de outubro de 1994, em outras palavras, é o CSS o responsável pela criação de páginas Web visualmente atraentes. Em uma analogia, o HTML seria o esqueleto da página, contendo todas as informações e elementos básicos enquanto o CSS é a ferramenta usada para dispor e personalizar a aparência desses elementos [17].

O DOM (Document Object Model) é uma convenção multiplataforma e independente de linguagem para representar e interagir com objetos em HTML [17]. De forma simplificada, é a representação dos elementos HTML em forma de árvore. A **Figura 05** abaixo mostra a representação do DOM de um trecho HTML:

[2] Nome que se dá às imagens ou palavras que dão acesso a outros conteúdos em um documento hipertexto. O hiperlink pode levar a outra parte do mesmo documento ou a outros documentos

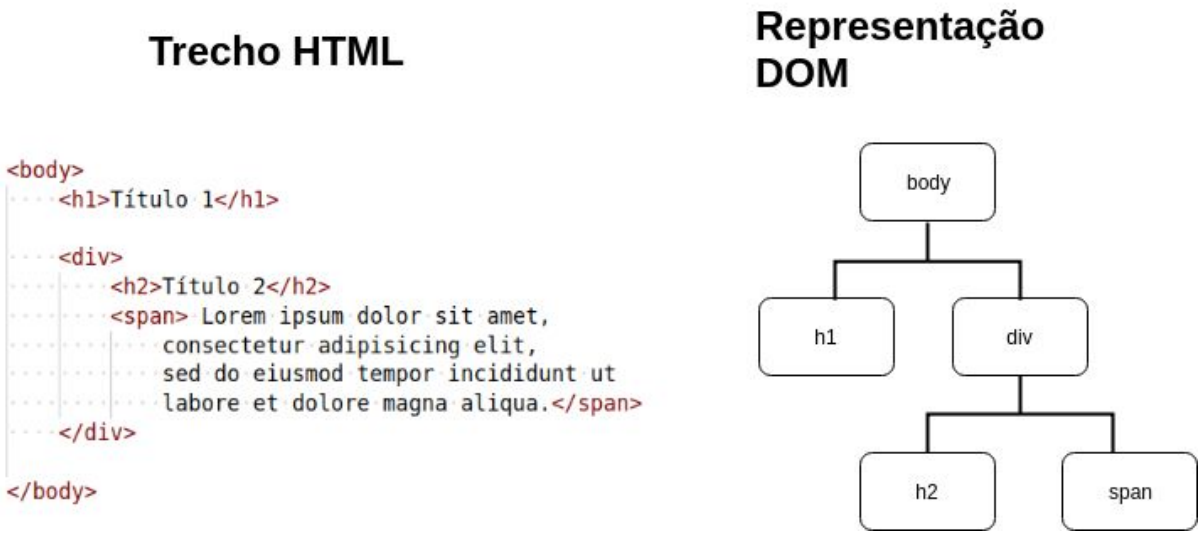


Figura 05 - Representação do DOM de um trecho HTML

2.3.3 Single Page Applications (SPAs)

O processo de desenvolvimento do ECMAScript facilitou o surgimento de uma nova abordagem de desenvolvimento para aplicações WEB: aplicações que eram de fato compostas por uma única página, cujo conteúdo era substituído à medida que processava as interações dos usuários [1], os chamados *Single Page Applications* (ou SPAs).

Fazer uma alteração no DOM (do inglês, *Document Object Model,* é a estrutura usada pelos navegadores para os elementos HTML em exibição na página) é muito custoso computacionalmente, em especial quando envolve uma grande quantidade de elementos. Isso se dá porque tais manipulações podem acarretar a necessidade de recalcular grande parte do *layout* da página. Em outras palavras, alterar um elemento no DOM, muitas vezes interfere em outros elementos da página [1].

Para contornar esse problema, normalmente o que alguns frameworks e bibliotecas fazem, é criar uma cópia do DOM, sem necessariamente ter relação com os elementos à mostra. Essa cópia é colocada na memória e, a partir dessa estrutura, computa-se o conjunto mínimo de alterações que devem ocorrer no DOM quando ocorre alguma alteração no estado da aplicação [1]. Por exemplo, o ReactJS, no qual será discutido mais à frente, faz uso de um algoritmo de diferenciação de árvores com o auxílio de heurísticas [18] para descobrir o que deve ser alterado na página. Essa estrutura tem o nome de virtual DOM.

Nos SPAs, parte do código é responsável por manter a interface em sincronia com os dados da aplicação. Para isso, existem alguns frameworks capazes de criar templates, facilitando assim, a componentização dos elementos da interface [1].

2.3.4 Anatomia De um SPA

Como em qualquer aplicação WEB, o ponto de entrada de um SPA é um arquivo HTML, mas as abordagens podem ser diferentes dependendo da biblioteca ou framework utilizado, como será visto na seção 2.3.5. Por exemplo, o KnockoutJS (seção 2.3.5.1) e o AngularJS (Seção 2.3.5.3) utilizam tanto HTML e JavaScript para criação da interface. Em contrapartida, no ReactJS (seção 2.3.5.2) o ponto de entrada é um arquivo HTML, mas diferentemente de outras abordagens, o conteúdo deste arquivo não há nenhum elemento de interface definido, apenas metadados e a importação de código JavaScript. É nesse arquivo JavaScript em que toda a aplicação está contida: geração de elementos de interface, lógica de negócios e comunicação com outros serviços [1].

A melhor maneira de construir uma interface de uma aplicação é através de uma componentização dos elementos da interface. Para isso, algumas bibliotecas e frameworks que serão abordados mais à frente podem auxiliar na criação destes templates. Comumente, os componentes são definidos declarativamente, e a apresentação e atualização dos elementos de interface que os compõem é resolvida pelo funcionamento interno da ferramenta de interface utilizada (framework ou biblioteca escolhida) [1].

Como dito na seção 2.3.3, mudanças no DOM podem ser altamente custosas do ponto de vista computacional, por isso abordagens como a do ReactJS são válidas.

Outro fator que deve ser levado em consideração quando se discute sobre interfaces de SPAs, é como se dá a ligação entre as camada de modelo e a camada de visão, ou seja, os dados da aplicação e a interface que apresenta esses dados. Pode ser feito através de uma **ligação de duas vias** ou de uma **ligação de via única**.

Na ligação de duas vias, como mostrado na **Figura 06**, os elementos da view e os de estado estão fortemente ligados. Ferramentas como KnockoutJS, VueJS e AngularJS implementam internamente entidades que observam mudanças tanto na interface quanto no estado da aplicação [1].

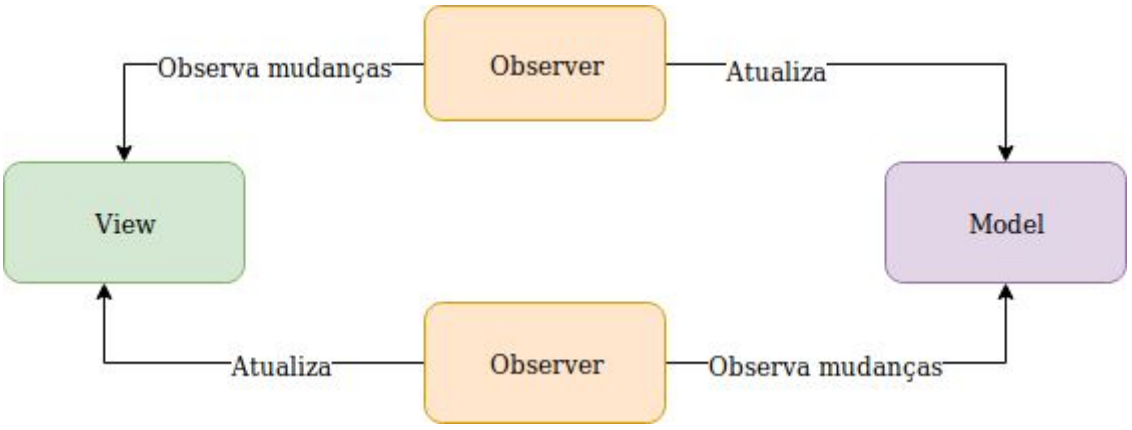


Figura 06 - Funcionamento de uma ligação de duas vias.

Adaptado de [19]

Na ligação de via única, como mostrado na **Figura 07**, os elementos de interface não são observados por mudanças, logo, as alterações de estado são feitas de forma explícita, através do tratamento de eventos disparados pela interação do usuário com a interface da aplicação. Atualmente, essa abordagem é preferível, pois torna o comportamento da aplicação mais previsível. O ReactJS utiliza esse tipo de abordagem.

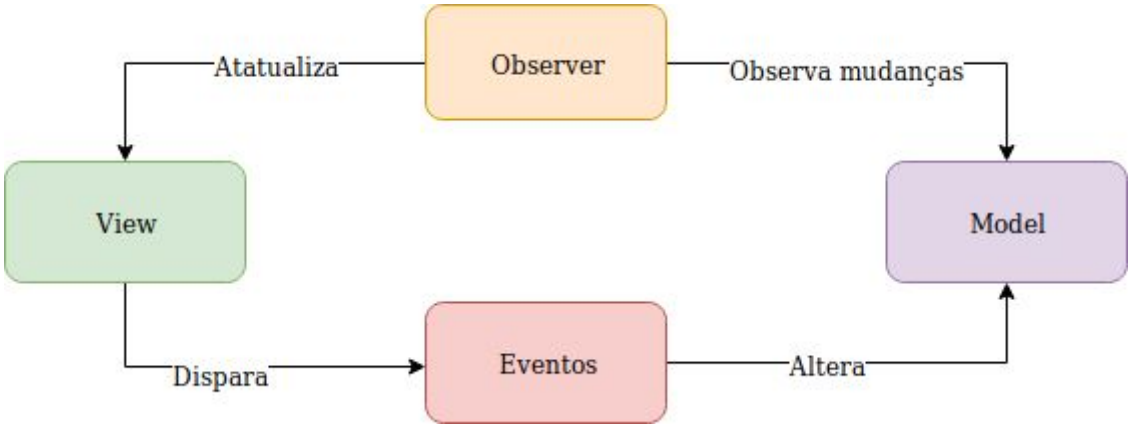


Figura 07 - funcionamento de uma ligação de uma via.

Adaptado de [19]

2.3.5 Frameworks e Bibliotecas

Framework nada mais é do que um conjunto de código que serve para facilitar a implementação de funcionalidades genéricas, possibilitando que o desenvolvedor se concentre em atividades mais complexas e pertencentes ao escopo da empresa [20], sendo assim, o programador se concentra em "o que" fazer, não em "como" fazer. Já bibliotecas em computação, como definido em [17], “oferecem uma série de funções pré-definidas úteis que você pode ‘chamar’ para melhorar e expandir a sua aplicação”.

Normalmente, o framework descreve a estrutura do aplicativo e mostra uma maneira de organizar seu código para tornar seu aplicativo mais flexível e escalável enquanto bibliotecas são mais flexíveis quanto ao seu uso.

A seguir, será descrito alguns dos principais frameworks e bibliotecas voltados para SPAs como o KnockoutJS, AngularJS, e ReactJS.

2.3.5.1 KnockoutJS

O KnockoutJS, ou knockout.js, ou ainda apenas Knockout (KO), é uma biblioteca que ajuda a criar interfaces ricas e responsivas com seus respectivos modelos de dados. Com o Knockout é fácil criar telas de interface dinâmicas (por exemplo, que mudem dependendo de uma ação do usuário ou de mudanças externas) [21]. Esta interface foi desenvolvida e mantida como código aberto por Steve Sanderson, funcionário da Microsoft.

Por seguir o padrão MVVM, os principais princípios do KO são:

- Uma clara separação entre os dados do domínio, ou seja, uma camada de modelo bem definida;
- A presença de uma camada bem definida de código especializado para gerenciar as relações entre os componentes da view, o chamado *viewmodel*.

Knockout é construído em torno de três recursos principais:

- ***Observables* e rastreamento de dependência**. Com isso, o KO consegue identificar quando uma parte do modelo teve mudanças.
- **Ligações declarativas**

- ***Templating***. Com isso, é possível reaproveitar componentes e evitar código repetido.

O código da **Figura 08** abaixo ilustra uma aplicação feita em KO que renderiza 3 listas e exibe o tempo levado.

```
39      // Knockout
40      function _knockout() {
41          // função de ativação do KO
42          ko.applyBindings({
43              // select recebe o item selecionado pelo usuário
44              selected: ko.observable(),
45              // data é o array observable
46              data: ko.observableArray(),
47              // função para selecionar um item
48              select: function(item) {
49                  this.selected(item.id);
50              },
51
52              // run chama a função count para gerar o array de objetos
53              run: function() {
54                  var data = count(),
55                      date = new Date();
56
57                  this.selected(null);
58                  // data: ko.observableArray() é chamado para se tornar o observable
59                  this.data(data);
60                  // quando terminar a renderização é calculado a diferença de tempo
61                  document.getElementById("run-knockout").innerHTML =
62                      (new Date() - date) + " ms";
63              }
64          }, document.getElementById("knockout"));
65      }
66
67      // função para detectar e responder a mudanças ao array
68      ko.observableArray.fn.reset = function(values) {
69          var array = this();
70          this.valueWillMutate();
71          // adiciona os itens ao array
72          ko.utils.arrayPushAll(array, values);
73          // notifica o array que houve mudanças
74          this.valueHasMutated();
75      };
76
```

```
39    // Knockout
40    function _knockout() {
41        // função de ativação do KO
42        ko.applyBindings({
43            // select recebe o item selecionado pelo usuário
44            selected: ko.observable(),
45            // data é o array observable
46            data: ko.observableArray(),
47            // função para selecionar um item
48            select: function(item) {
49                this.selected(item.id);
50            },
51
52            // run chama a função count para gerar o array de objetos
53            run: function() {
54                var data = count(),
55                    date = new Date();
56
57                this.selected(null);
58                // data: ko.observableArray() é chamado para se tornar o observable
59                this.data(data);
60                // quando terminar a renderização é calculado a diferença de tempo
61                document.getElementById("run-knockout").innerHTML =
62                    (new Date() - date) + " ms";
63            }
64        }, document.getElementById("knockout"));
65    }
66
67    // função para detectar e responder a mudanças ao array
68    ko.observableArray.fn.reset = function(values) {
69        var array = this();
70        this.valueWillMutate();
71        // adiciona os itens ao array
72        ko.utils.arrayPushAll(array, values);
73        // notifica o array que houve mudanças
74        this.valueHasMutated();
75    };
76
```

Figura 08 - Exemplo de código em KO [17]

2.3.5.2 ReactJS

O reactJS, ou react.js é uma biblioteca JavaScript criada pelo facebook mas, diferentemente dos outros frameworks que serão tratados, no qual fornecem todos os componentes necessários para uma aplicação front-end, o React se preocupa apenas com a parte da view [17].

Uma de suas principais características é o uso do “virtual DOM”. Como dito anteriormente, realizar uma mudança na camada de visão é altamente custoso, tendo em vista que para alterar o DOM, é necessário fazer cálculos de reposicionamento de toda a página.

Mas no react, uma cópia do DOM é gerada na memória do computador, e é essa cópia que será modificada. O DOM real é então gerado a partir desta representação.

No react, o gerenciamento de dados é uma tarefa inerentemente complexa. Para uniformizar, a forma com que os dados são acessados na aplicação, o Facebook propôs o padrão arquitetural Flux, cujo o fluxo de dados é unidirecional, como explicado na seção 2.3.4. A **Figura 09** abaixo, ilustra a arquitetura proposta.

Nesta arquitetura, as ações são encapsuladas em *dispacthes*. Estes delegam o processamento para as *stories* (objetos que contém partes do estado da aplicação). Quando um atributo dentro destes objetos é modificado, os elementos da *view* são notificados por meio o padrão *observable*, e então dispara novas ações em reação às interações do usuário [22].

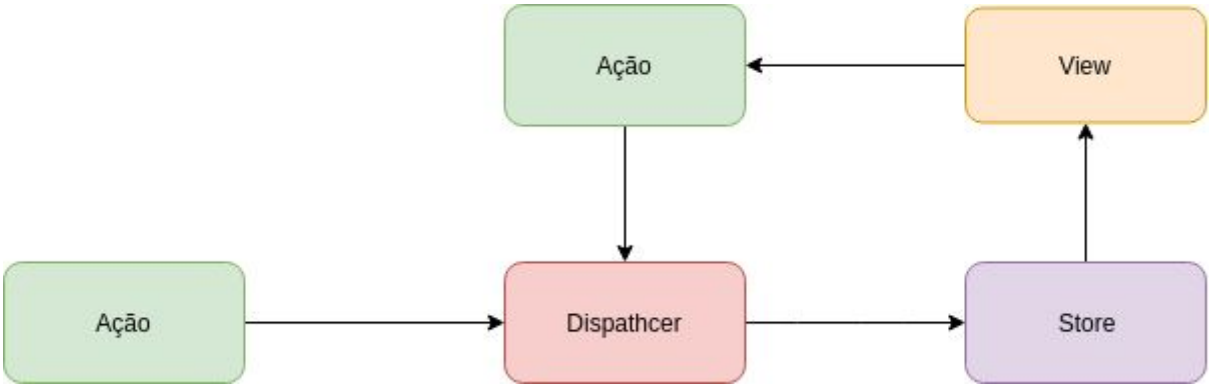


Figura 09 - Padrão Flux do React

Existem também soluções que são adaptações do Flux, como o Redux [23]. Essa biblioteca auxiliar tem a intenção de simplificar os conceitos do Flux, como a existência de múltiplas Stores e a implementação de alterações de estado por meio de funções puras [1].

O código da **Figura 10** abaixo ilustra a mesma aplicação feito em React.

```
// React
function _react() {
    // definição de classe
    var Class = React.createClass({
        // função para selecionar um item
        select: function(data) {
            this.props.selected = data.id;
            this.forceUpdate();
        },

        // função para o react renderizar o objeto e manipular diretamente o html
        render: function() {
            var items = [];

            // react interage diretamente com os elementos e classes html
            for (var i = 0; i < this.props.data.length; i++) {
                items.push(React.createElement("div", { className: "row" },
                    React.createElement("div", { className: "col-md-12 test-data" },
                        React.createElement("span", {
                            className: this.props.selected ===
                                this.props.data[i].id ? "selected" : "",
                            onClick: this.select.bind(null, this.props.data[i]) },
                                this.props.data[i].label)
                    )
                ));
            }

            return React.createElement("div", null, items);
        }
    });

    var runReact = document.getElementById("run-react");
    // evento para chamar a criação do objeto e renderizar o objeto
    runReact.addEventListener("click", function() {
        var data = run(),
            date = new Date();

        React.render(new Class({ data: data, selected: null }),
            document.getElementById("react"));

        runReact.innerHTML = (new Date() - date) + " ms";
    });
}
```

```
<!-- REACT -->
<div class="col-md-3 band">
    <div class="row">
        <div class="col-md-7">
            <h3>React</h3>
        </div>
        <div class="col-md-5 text-right time" id="run-react">Iniciar</div>
    </div>
    <div id="react"></div>
</div>
```

Figura 10 - Exemplo de código em React [17]

2.3.5.3 AngularJS

O AngularJS, ou angula.js, ou apenas angular, é um framework de código aberto mantido pela Google e pela comunidade. Por seguir o padrão MVC, ele é capaz de manter o esforço de desenvolvimento, manutenção e testes mais fáceis [24].

Ao contrário do React em que a view é feita em JSX, aplicações SPAs que usam AngularJS e o Knockout, têm a sua view implementadas em HTML [24].

O AngularJS introduz uma tag adicional na view, chamada de "diretivas". Essas diretivas têm o prefixo "ng-" e assim como no KO, essa diretiva realiza um bind de duas vias entre view (ou templates) e o controller [24].

Segundo JADHAV (2015), pode-se citar as seguintes vantagens ao usar este framework:

- **MVC:** O AngularJS utiliza o padrão MVC, e como foi visto na seção 2.2.1, são feitas 3 camadas, uma para a view, uma para a lógica, e uma para os modelos. Essa divisão ajuda a facilitar o desenvolvimento
- **Ligação de duas vias:** como foi visto na seção 2.3.4, os dados são atualizados sejam as mudanças ocorrendo na view ou no modelo, ou seja, a view atualiza o modelo e vice-versa.
- **Facilidade em desenvolver a UI:** Já que o angular usa o padrão MVC, é possível criar e desenvolver a interface sem depender da lógica atrás da view

O código da **Figura 11** ilustra a mesma aplicação feito em Angular.

```
2  <!DOCTYPE html>
3
4  <html ng-app="app">
5      <head>
```

```
<!-- ANGULAR -->
<div class="col-md-3 band2">
    <div class="row">
        <div class="col-md-7">
            <h3>Angular</h3>
        </div>
        <div class="col-md-5 text-right time" id="run-angular" ng-click="initialize()">
            Iniciar
        </div>
    </div>
    <div>
        <div class="row" ng-repeat="item in data">
            <div class="col-md-12 test-data">
                <span ng-class="{ selected: item.id === $parent.selected }"
                    ng-click="select(item)"> {{item.label}}
                </span>
            </div>
        </div>
    </div>
</div>
```

```
// Angular
angular.module('app', []).controller('controller', function($scope) {

    $scope.initialize = function() {
        var data = count(),
            date = new Date();

        $scope.selected = null;
        // é disparado quando um ciclo do Angular é completado
        $scope.$$postDigest(function() {
            document.getElementById('run-angular').innerHTML =
                (new Date() - date) + ' ms';
        });

        $scope.data = data;
    };

    // função para selecionar um item
    $scope.select = function(item) {
        $scope.selected = item.id;
    };
```

Figura 11 - Exemplo de código em Angular [17]

2.4 Banco De Dados

A fim de aumentar a eficiência com que os dados de uma empresa são gerenciados, diminuir os custos e centralizar o conhecimento do sistema, o uso de um banco de dados é crucial para qualquer sistema de informação. Assim, diversas técnicas de análise de dados podem ser aplicadas para ajudar na tomada de decisão [25][26].

Por isso a escolha do banco de dados é muito importante. Existem hoje no mercado vários tipos e cada um deles serve a diferentes propósitos.

2.4.1 Relacional

Segundo SILBERSCHATZ (2011), o modelo de banco de dados relacional é baseado em tabelas e armazena tanto os dados quanto as relações entre eles. Para que os usuários pudessem interagir com o banco, foi criada uma linguagem específica, o SQL (do inglês, Structured Query Language) [27].

Com o SQL é possível:

- Manipular a estrutura de cada tabela;
- Manipular as instâncias de cada tabela;
- Definir restrições de integridade que os dados contidos nas tabelas devem seguir;
- Controlar o começo e o fim de cada bloco, assim é possível descartar todos os comandos do bloco, caso algum erro ocorra;
- Definir permissões de acesso a tabelas e a visões;
- Permite que linguagens de programação de propósito geral como Java ou C++ por exemplo, possam executar código SQL utilizando sua sintaxe nativa.

Uma das principais características dos bancos relacionais são as propriedades conhecidas pelo acrônimo ACID, como listadas em [33]:

- **Atomicidade:** é possível colocar várias operações em uma única transação, e se alguma operação der erro, todas as outras serão descartadas, ou seja, ou todas são executadas, ou nenhuma.
- **Consistência:** Após uma transação ter sido concluída, o banco de dados deve permanecer em um estado consistente, ou seja, deve satisfazer as condições de consistência e restrições de integridade previamente assumidas.
- **Isolamento:** Uma transação não sofre interferência de outras transações.
- **Durabilidade:** Uma vez que uma transação ocorra com sucesso, seu efeito não poderá ser desfeito.

Vale ressaltar que, apesar da simplicidade, é muito importante que o modelo de dados passe por uma normatização[3] antes de ser construído efetivamente, para evitar problemas de redundância e performance [25].

2.4.2 Orientados a Objeto

O primeiro obstáculo que os programadores tiveram que encarar usando o modelo relacional da seção acima, é a limitação dos tipos de dados que podem ser armazenados. Aplicações complexas requerem dados mais complexos, dados aninhados, atributos multivalorados e herança. Essas características essas, que são fornecidas por linguagens de programação baseadas em orientação a objeto [25].

O segundo obstáculo é a dificuldade de acessar os dados de um banco relacional através de uma linguagem de programação como Java ou C++. Simplesmente estender os tipos suportado pelo banco não é o suficiente. O que o banco orientado a objetos propõe é retirar a linguagem intermediária (SQL) pois, segundo SILBERSCHATZ (2011), deixa o trabalho do programador mais difícil. Assim o acesso se dá por meio de extensões.

2.4.3 NoSQL

Ao contrário do que muitos pensam o noSQL não significa "ausência" ou "rejeição" da abordagem definida pelos bancos relacionais. Vem do inglês *not-only* SQL.

Com o passar do tempo, os sistemas estavam ficando cada vez mais complexos e processando uma quantidade maior de dados, e o tempo de resposta dos bancos tradicionais não estava satisfazendo as necessidades. Foi daí que um novo paradigma surgiu – Os bancos NoSQL [28]

A forma que os sistemas tradicionais têm de escalar é verticalmente, ou seja, é feita adicionando mais poder de processamento e armazenamento em um único cluster – o que torna o processo muito caro [28]. Mas, diferentemente da abordagem dos bancos de dados relacionais, os bancos NoSQL, não armazena os dados em tabelas, permitindo que esses

[3] Conjunto de regras e boas práticas que devem ser levadas em consideração no momento da elaboração do banco de dados

sistemas fossem escalados horizontalmente, utilizando-se de vários clusters (muitas vezes de pequeno porte) [28], o que é muito mais barato.

Os sistemas NoSQL se dividem em quatro grandes categorias principais:

- **Chave-valor**: Assim como em qualquer arquivo JSON, o dado é salvo mapeando um valor para uma chave. Cada chave deve ser única e corresponde a um valor, que pode representar desde um simples valor inteiro até agregações complexas de dados [27]. Um exemplo de SGBD que implementa esse tipo de modelo é o REDIS [29].
- **Baseada em Documento**: é baseado na mesma ideia de chave-valor, mas além dos valores, também permite que sejam armazenados dados auxiliares que facilitam a manipulação de dados, os meta-dados [27]. O mongoDB se destaca quando se fala desse modelo de banco de dados [30].
- **Família de colunas**: assim como no modelo relacional, a estrutura interna de armazenamento é baseado em tabelas, embora não possam ser aplicados os mesmos conceitos. É ideal para grandes quantidades de dados [27]. O SGBD mais utilizado neste modelo é o Cassandra [31].
- **Orientada a Grafos**: baseada na teoria dos grafos, é ideal para modelar dados que possuem um alto grau de conectividade entre si como por exemplo, redes de computadores e sistemas de rede social [27]. O principal SGBD desse modelo é o Neo4j [32].

2.4.4 NewSQL

Neste modelo de banco de dados, a ideia é agregar as características dos modelos noSQL aos bancos de dados relacionais, ou seja, estender os benefícios do modelo relacional para arquiteturas distribuídas ou agregar as propriedades ACID ao modelo NoSQL [33].

A principal vantagem do NewSQL é manter a linguagem SQL, o modelo relacional e ao mesmo tempo agregar as principais vantagens do NoSQL, o que possibilita aos desenvolvedores oriundos do mercado tradicional SGBD suavizarem a curva de aprendizado e

adaptação à nova tecnologia. Sua principal desvantagem é a falta de maturidade das ferramentas existentes no mercado, e a pequena quantidade delas [33].

O banco de dados VoltDB, por exemplo, é um banco relacional com características do modelo não relacional. Ele utiliza das transações ACID do modelo relacional, e ao mesmo tempo possui alta escalabilidade e disponibilidade do modelo não-relacional [33].

Já o FoundationDB é um banco não-relacional com características do relacional. Por exemplo, ele tem suporte para transações ACID do relacional ao mesmo tempo que é horizontalmente escalável, característica de modelos não-relacionais. [33].

3. Sobre o Sistema

Esse capítulo fornece uma visão do sistema cuja a análise foi realizada para identificar oportunidades de reengenharia a fim de melhorar propriedades como testabilidade, desempenho e remoção de código morto (fora de uso). A análise teve como base o entendimento do sistema a partir de seu funcionamento e estudo do código fonte. Dessa análise foram extraídos os principais requisitos funcionais e não funcionais e casos de uso que estão detalhados no documento de requisitos (**APÊNDICE A – DOCUMENTO DE REQUISITOS DO SISTEMA POS/ERP DA EMPRESA X**).

3.1 Breve Descrição do Sistema

A empresa X é especializada no setor têxtil - roupas sob medida. A empresa possui diversas etapas a serem executadas durante o seu ciclo básico de trabalho. Passando pelos processos de venda, produção, entrega e acompanhamento do produto, controle de qualidade, acabamento e logística.

Existem dois principais softwares dentro da empresa, conhecidos pelos acrônimos POS e ERP.

O POS (a parte mobile do sistema) e atua principalmente com a venda inicial do produto e o acompanhamento após a produção. Esse software é utilizado em tablets apenas pelos vendedores que têm contato direto com os clientes. Os vendedores o utilizam para captar dados dos clientes, oferecer os produtos disponíveis e captar o pedido final com todos os produtos e seus detalhes. Com ele é possível também acompanhar a logística do produto.

Ao final dos processos de análise de pedido, produção e logística (realizados pelo ERP), o vendedor utilizar o POS novamente para acompanhar o produto final, verificando se é necessário ou não ajustes finos.

O ERP atua principalmente com o manejo e ajustes internos dos pedidos. Esse software é somente utilizados por administradores e alfaiates, geralmente com habilidades bastante refinadas nesse tipo de negócio. Eles analisam cada pedido minuciosamente, desde

medidas até cada opção de design escolhida pelos clientes. Além da análise inicial do pedido, os administradores acompanham todo o processo de produção e logística dos produtos, visto que a empresa atua em diversos territórios e países. Após a submissão do pedido, os administradores revisam todos os detalhes e aprovam para a produção. Após a entrega, o ERP é utilizado novamente para acompanhar ajustes finos nos produtos finais, se necessário.

O sistema também é responsável pela análise dos dados da companhia como a quantidade de peças que um determinado funcionário vendeu; o lucro total em um ano; qual peça, e design são mais lucrativos; logística; perfil dos usuários, entre outros fatores.

3.2 Usuários do Sistema

Com base no entendimento do negócio da empresa através da análise de código do sistema X, foi identificado quatro tipos de perfis de usuário, são eles: vendedor, alfaiate, administrador e cliente.

- **Vendedor:** Usuário do POS, ele é responsável por atender o cliente em um dos pontos de vendas e realizar o processo de venda, ou de "*fitting*" (processo de auxiliar o cliente a provar a peça e se ela atende aos requisitos do cliente)
- **Alfaiate:** Ele baixa um PDF do ERP e a partir dele, o alfaiate monta a peça de acordo com as requisições do cliente. Se o cliente possuir alguma queixa durante o *fitting*, a peça volta para a fábrica e o alfaiate faz os ajustes necessários
- **Administrador:** Possui acesso total ao sistema, tanto no POS como no ERP. Ele pode acompanhar os pedidos e realizar um novo. Ele é o único que pode aprovar um pedido depois que ele é feito.
- **Cliente:** Não atua diretamente em nenhum dos sistemas, exceto no ato do pagamento, onde ele insere informações como método de pagamento e sua assinatura.

3.3 Arquitetura

A arquitetura do sistema POS é baseada no sistema MVVM (do inglês, *Model-View-ViewModel*) descrito na seção 2.3.5.1. A View (camada de visão) através do

data-binding, é conectada com a ViewModel notificando a ocorrência de eventos. A ViewModel por sua vez, responde a esta notificação realizando alguma ação no modelo, como por exemplo, obtendo algum dado, atualizando ou inserindo informações no banco de dados [34].

Já a arquitetura do ERP é baseada no modelo MVC (do inglês, Model-View-Controller). A camada de visão do sistema interage diretamente com os usuários, captando entradas e exibindo resultados. Através de requisições ajax e usando o protocolo HTTP, os usuários se comunicam com a camada dos controladores. Essa camada então, utiliza diversos modelos para executar a lógica de negócio e retornar os resultados para os usuários. É a camada de modelos quem manipula o banco de dados [27].

Algumas bibliotecas auxiliares também são utilizadas no sistema, são elas tagdd, raphael.js, FPDF e FFMPEG do lado do ERP e do lado do POS é usado Jquery, JqueryUI, bootstrap e o KnockoutJS.

Jquery, como introduzido na seção 2.3.1, é uma das bibliotecas mais usadas em javascript. Com ela é possível que o código funcione da mesma forma nos diferentes navegadores, pois ela abstrai várias funcionalidades nativas do Javascript [35].

A JqueryUI, é uma biblioteca da camada de visão que possui vários elementos visuais já prontos, e que podem simplesmente ser importados ao sistema [36].

O bootstrap é a mais famosa biblioteca *front-end* do mercado. Ela contém elementos prontos para serem usados por diferentes tamanhos de tela, permitindo uma maior consistência e estabilidade em praticamente qualquer dispositivo [37].

A biblioteca Taggd.js é uma ferramenta usada para marcar elementos em ilustrações. Com ela é possível inserir etiquetas e a sua descrição em uma imagem. A biblioteca é utilizada em diversas partes do sistema, já que seria praticamente impossível realizar todos os processos de fabricação e correções dos produtos sem auxílio visual [38].

A biblioteca *RaphaelJS* é uma ferramenta de desenho que permite com que os alfaiates desenhem diretamente em uma imagem quais alterações foram feitas nos painéis corporais dos clientes [39].

A biblioteca FFPDF é responsável pela elaboração e PDFs. São arquivos nesse formato que são enviados entre os diferentes setores da empresa desde a produção de um

produto à elaboração de relatórios. A biblioteca é codificada em PHP e é executada pelos controladores e modelos do sistema [40].

O FFMPEG é na verdade um software que possui diversas bibliotecas para manipulação de vídeo e áudio [41].

O KnockoutJS, como já foi abordado, é uma biblioteca que ajuda a criar interfaces de usuário ricas e de visualização responsiva. Com ele é fácil atualizar uma parte específica da interface dinamicamente, por exemplo, por meio de ações do usuário ou quando uma fonte de dados externa for alterada [21].

3.4 Arquitetura POS/ERP

O POS utiliza como principal ferramenta o KnockoutJS, e por esse motivo ele é guiado pelo padrão MVVM de forma que existe uma espécie de fachada entre as classes de modelo e a visão [34]. Conforme mostrado no fluxograma abaixo:

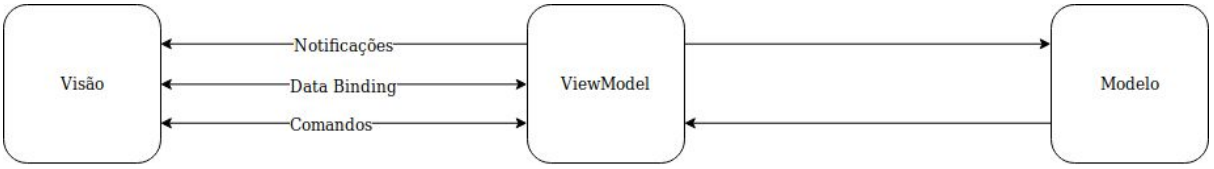


Figura 12 - Modelo simplificado do MVVM

As classes de modelo, como no conceito do padrão do MVC, são aquelas responsáveis pela lógica do negócio da empresa.

As classes de visão são aquelas responsáveis por exibir os elementos na tela do dispositivo.

As classes conhecidas como *viewModel* são aquelas que intermediam as duas classes anteriores. Elas contém uma lógica de apresentação e provê os dados para a camada de visão através do data-bind, comandos e notificações [27].

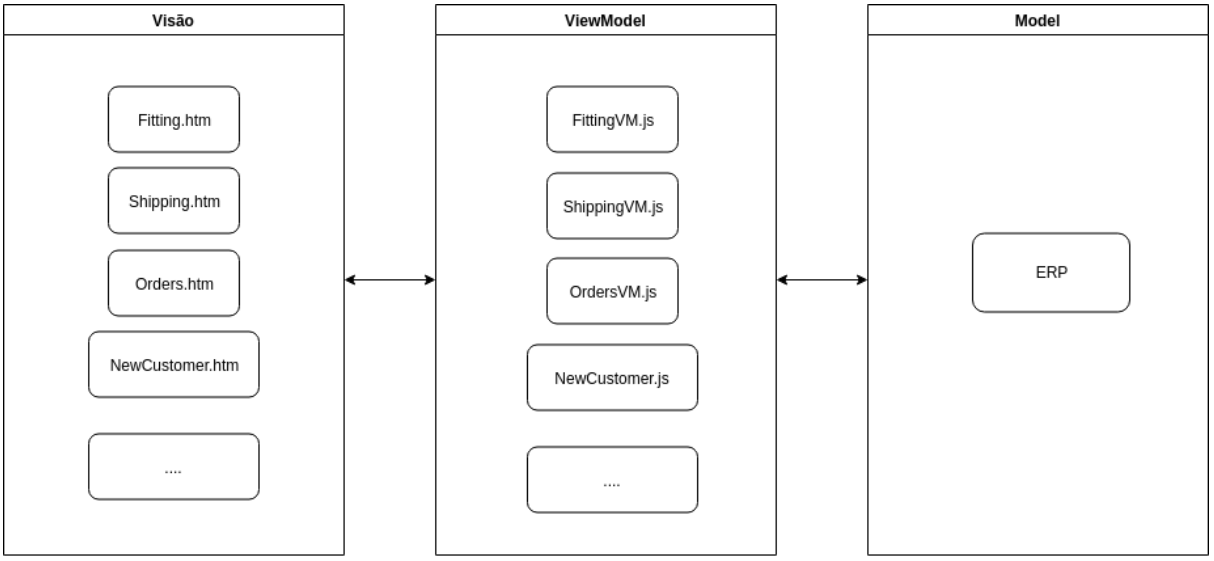


Figura 13 - Modelo simplificado do POS

No diagrama acima, é possível ver que as classes de modelo não são encontradas no POS, e sim no ERP, pois é ele quem se comunica com o banco para só então enviar os dados para os *viewModels* do POS.

4. Proposta de melhorias

4.1 Arquitetura

Apesar de todas as arquiteturas apresentadas neste trabalho apresentarem variações em seus detalhes, seus objetivos tendem a se alinhar: Dividir o software em camadas bem definidas e facilitar a testabilidade.

As características que fazem o MVC, um dos modelos mais simples de aplicar, mais utilizados e que exigem uma menor quantidade de código, trazem certos problemas à medida que o sistema vai crescendo e ficando mais complexo. Operações relacionadas a persistência ou networking não são vistas como preocupações do padrão MVC, que foi concebido para resolver problemas de interações via interface do usuário [42]. Ainda em detrimento da facilidade no começo do desenvolvimento, há impactos negativos na manutenção e testabilidade do código, tornando o desenvolvimento propenso a diversos problemas. Ainda assim, é a melhor opção para projetos pequenos que não exigem arquiteturas mais elaboradas e que não têm a intenção de se preocupar com um processo de manutenção mais rigoroso [42]. Como não é o caso do sistema da empresa X, o MVC não é recomendado para a substituição dos softwares atuais.

Entre os modelos arquiteturais analisados restantes, o MVP e o MVVM, é realmente difícil escolher qual deles é o melhor. Muitos fatores influenciam nessa decisão, entre eles os frameworks usados. Por exemplo, caso do Knockout seja mantido, a arquitetura que deve ser seguida é o MVVM. Caso o React seja escolhido, a arquitetura que deve ser seguida é o padrão Flux ou uma adaptação dele, o Redux. Outros frameworks podem ser mais flexíveis ou requerem outros modelos arquiteturais.

4.2 Frameworks e Bibliotecas

No artigo “What leads developers towards the choice of a JavaScript framework?”, (em uma tradução livre significa “O que leva os desenvolvedores a escolher um framework JavaScript”) por Pano, Graziotin e Abrahamsson (2016) [43], foram entrevistados 18 desenvolvedores responsáveis pela tomada de decisão na escolha de um framework JavaScript de suas respectivas empresas. Eles relatam que existem algumas características básicas que a aplicação deve possuir tais como: a vinculação de dados, a manipulação de DOM e a atualização em tempo real da aplicação [20]. Além disso, é muito importante que a biblioteca ou framework permitam modificações de qualquer tipo através de operações simples e que as estruturas sejam modulares, ou seja, se uma parte do código for alterada, ela não comprometa a aplicação geral.

Neste experimento foram listadas as seguintes características aos entrevistados:

- Usabilidade, ou seja, o quão fácil de usar o framework é;
- Custo;
- Eficiência, ou seja, desempenho e tamanho da aplicação final, levando em consideração a quantidade de código necessária, tempo de renderização (carregamento da página);
- Funcionalidade, ou seja, o que a biblioteca ou framework provê em relação a automatização, extensibilidade, flexibilidade, isolamento, modularidade e atualização;

Neste experimento, todos os participantes mencionaram eficiência como o principal fator de influência na escolha de um framework JavaScript.

Em [20], foi feito um estudo comparando o desempenho de diferentes frameworks e bibliotecas JavaScript. Nesta monografia, o tempo de execução foi apurado nos três principais browsers da atualidade, o Chrome, Firefox e o Safari. O algoritmo utilizado no teste gera um *array* de 1000 objetos em que é calculado o tempo desde o início da geração do *array* até a visualização do usuário no browser.

Neste estudo, como ilustrado nas **Figuras 14**, **15**, e **16**, foi constatado que o React tem uma menor variação de acordo com a média nos três browsers testados, além de ter uma menor média de tempo de execução. Isso se deve ao fato de que o React é o único que não manipula o DOM diretamente, e sim com o auxílio de um DOM virtual. O DOM virtual evita mudanças no DOM, que são caras em termos de desempenho, porque causam uma nova renderização da página e, por conseguinte do DOM. O DOM virtual também permite coletar várias alterações a serem aplicadas de uma só vez [20].

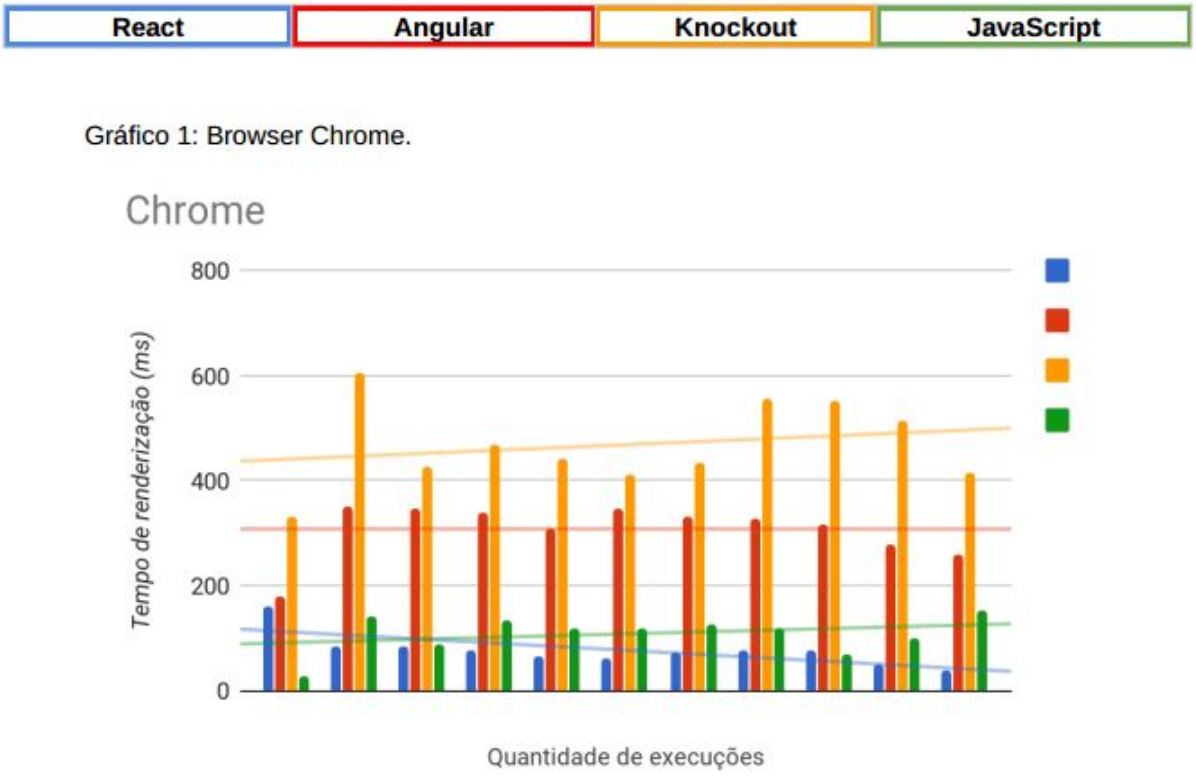


Figura 14 - Gráfico de desempenho das bibliotecas estudadas no Chrome [20]

Figura 15 - Gráfico de desempenho das bibliotecas estudadas no Firefox [20]

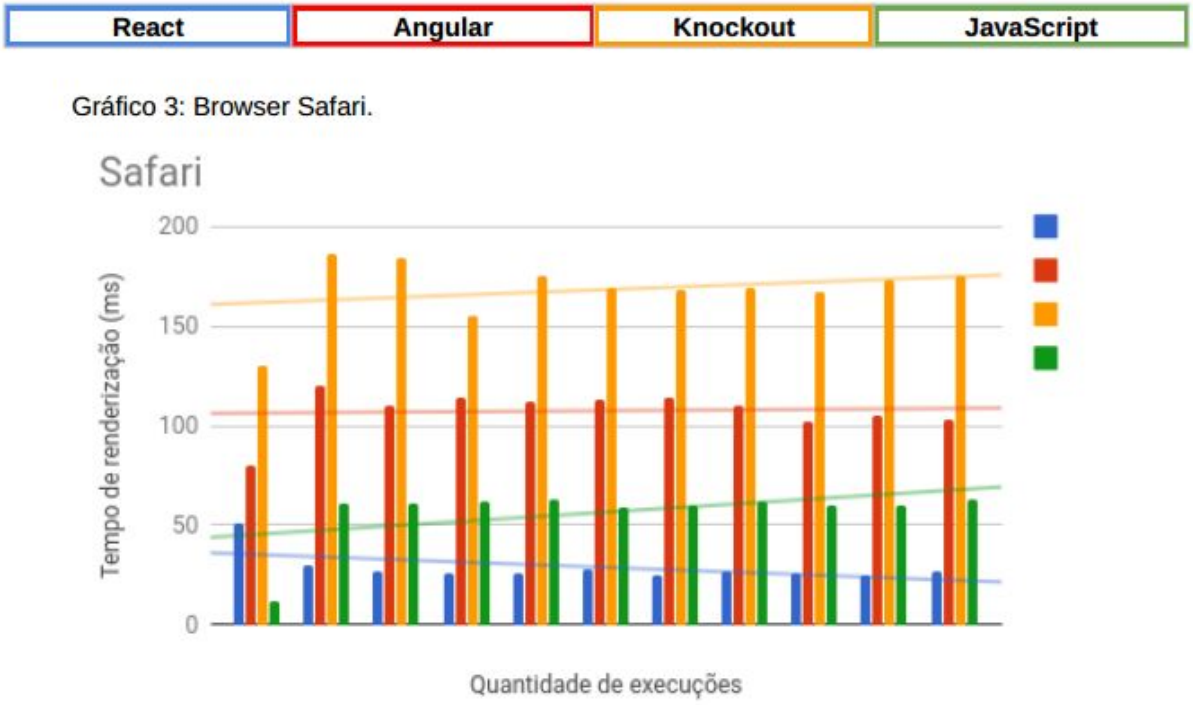


Figura 16 - Gráfico de desempenho das bibliotecas estudadas no Safari [20]

Segundo CECHINEL (2017) ainda, outro fator que influencia no desempenho de uma aplicação WEB é a quantidade de código a ser carregado pelo *browser*. O Angular é um framework, e por isso ele dita uma estrutura a ser seguida, estrutura essa que irá fornecer funções e callbacks ao desenvolvedor. Assim, o Angular possui o maior tamanho entre os produtos testados.

Já o React e o Knockout, por serem bibliotecas (coleções de classes e métodos que podem ser reutilizáveis) resultam em códigos menores.

Com JavaScript puro não há a necessidade de adicionar código extra para a aplicação, já que os browsers interpretam JavaScript nativamente. Porém, a desvantagem de usar somente JavaScript para o desenvolvimento web está na menor reutilização de código, por isso, a quantidade de código necessário para uma aplicação tende a ser maior [20].

Dito isto, apesar do Knockout atender às atuais necessidades do projeto, o POS poderia ter mais desempenho, se fosse feito em React.

4.3 Modelo de Banco de Dados

No quesito à escolha de banco de dados, quem define a melhor escolha é a natureza da aplicação. Não existe um modelo melhor do que o outro, e sim, casos em que um se aplica melhor. Não adianta querer usar um banco relacional para tratar de uma mídia social e esperar os mesmos resultados de um banco NoSQL. Em contrapartida, não adianta esperar que um sistema de e-commerce com banco NoSQL tenha o mesmo nível de segurança e confiabilidade dos bancos relacionais.

Na seção 2.4 pode-se perceber então que cada um desses modelos são melhores aplicados em determinadas situações. O modelo relacional é mais indicado para ambientes onde a consistência e integridade dos dados é primordial. O modelo NoSQL tem maior desempenho na distribuição de dados e abre mão da consistência rígida do modelo ACID, sendo mais indicado para cenários com alta demanda por escalabilidade e desempenho sem alto nível de consistência. O NewSQL pretende juntar características tanto do NoSQL como no modelo relacional, sendo capaz de ter as características do ACID e desempenho em processamentos de dados distribuídos, porém, ele ainda é incapaz de manter 100% de

consistência em suas transações, característica do modelo relacional. A **tabela 01**, ilustra melhor as diferenças entre os modelos de banco de dados citados.

O banco de dados orientado a objeto, é recomendado para aplicações cujo os dados a serem guardados sejam complexos, a ponto de necessitar dos paradigmas de uma programação orientada a objeto, bem como herança, polimorfismo e encapsulamento [46]. Embora alguns pontos do sistema da empresa X sejam realmente complexos, o banco de dados relacional usado está suprindo todas as necessidades, e o custo dessa migração é maior do que o possível ganho.

Tabela 01 - Comparativo entre os modelos de Banco de Dados [33]

	Relacional	NoSQL	NewSQL
Esquema	Rígido	Flexível	Flexível
Tran. ACID	Sim	Não	Sim
A – Atomicidade	Sim	Não	Sim
C – Consistência	Sim	Não	Não
I – Isolamento	Sim	Sim	Sim
D - Durabilidade	Sim	Não	Sim
CAP	CA	AP	CAP
Replicação	Sim	Sim	Sim
Part. Vertical	Sim	Não	Não
Part. Horizontal	Não	Sim	Sim
Sharding	Não	Sim	Sim

Dito isto, como o sistema da Empresa X possui dados bem estruturados, com pouca demanda e com mais consultas do que inserções, e que preza pela consistência e integridade, a decisão de usar um banco relacional foi uma boa escolha.

5. Conclusões e Trabalhos Futuros

Este trabalho realizou um estudo comparativo entre algumas tecnologias e ferramentas disponíveis para aplicações WEB, como arquitetura, modelo de banco de dados e frameworks. Além disso, foi feita uma engenharia de requisitos e projeto de software para os sistema POS/ERP.

Este trabalho realizou uma engenharia de requisitos e projeto de software para os sistemas POS/ERP e, também, um estudo comparativo entre algumas tecnologias e ferramentas disponíveis para aplicações WEB, como arquitetura, modelo de banco de dados e frameworks.

Foi constatado que para o modelo de banco de dados, a melhor alternativa para o e-commerce em questão, é o modelo relacional. Para o sistema do POS, a biblioteca mais recomendada, por motivos de desempenho, é o React. Por fim, quanto a escolha do modelo arquitetural, não foi possível escolher, uma vez que existem vários fatores que influenciam essa escolha.

Este estudo fica apenas como uma proposta, e como trabalho futuro, é sugerido a implementação de tais ferramentas e a comparação quantitativa dos dados. Uma definição mais detalhada de quais cenários seriam implementados e um planejamento das atividades de migração também deveria ser considerado.

6. Referências

[1] OLIVEIRA, Daniel de Jesus. Uma proposta de arquitetura para Single-Page Applications: Trabalho de Graduação. 2017. 47 f. Monografia (Especialização) - Curso de Ciência da Computação, Centro de Informática, Universidade Federal de Pernambuco, Recife, 2017.

[2] CHIKOFSKY, J. E., Cross, J. H. Reverse Engineering and Design Recovery: A Taxonomy. IEEE Software, v.7, n.1, pp. 13-17, 1990.

[3] DE JESUS, Elisângela Sato; FUKUDA, Ana Paula; DO PRADO, Antonio Francisco. Reengenharia de Software para Plataformas Distribuídas Orientadas a Objetos. 1999.

[4] Software Reengineering Technique Classification. Disponível em: <http://www.baiengineering.com/srtclass.html#771613> Acesso em: 10 de abril de 2019

[5] Introdução à Refatoração. Disponível em: <https://www.devmedia.com.br/introducao-a-refatoracao/21377> Acesso em: 9 de março de 2019

[6] SOMMERVILLE, IAN Software Engineering. 9. ed. 2011

[7] PINTO, Herbert Laroca Mendes; BRAGA, José Luís. Sistemas legados e as novas tecnologias: técnicas de integração e estudo de caso. Informática Pública, Belo Horizonte, v. 7, n. 1, p. 48-69, 2004.

[8] LUQUE, Leandro; VERISCIMO, Érico; PEREIRA, Girdácio. Entendendo e Estendendo o Framework MVVM Android Binding.

[9] Model-View-Controller (MVC). Disponível em:

<http://www.dsc.ufcg.edu.br/~jacques/cursos/map/html/arqu/mvc/mvc.htm>. Acesso em: Acesso em: 9 de março de 2019

[10] Android MVC x MVP x MVVM qual Pattern utilizar — Parte 1. Disponível em: <https://medium.com/@FilipeFNunes/android-mvc-x-mvp-x-mvvm-qual-pattern-utilizar-parte-1-3defc5c89afd>. Acesso em: 9 de março de 2019

[11] FOWLER, MARTIN. GUI Architectures. martinfowler.com, 2006. Disponível em: <https://www.martinfowler.com/eaaDev/uiArchs.html>. Acesso em: 9 de março de 2019.

[12] GAROFALO, R. Building enterprise applications with Windows Presentation Foundation and the model view viewmodel pattern. Sebastopol, Calif.: O'Reilly Media, Inc., 2011.

[13] History of the web - world wide web foundation. Disponível em: <https://webfoundation.org/about/vision/history-of-the-web/> Acesso em: 9 de março de 2019

[14] A short history of javascript. Disponível em: <https://www.w3.org/community/webed/wiki/A_Short_History_of_JavaScrip> Acesso em: 12 de março de 2019

[15] jquery. Disponível em: <https://jquery.com/> Acesso em: 10 de março de 2019

[16] B. Eich. (2008, aug) Ecmascript harmony. Disponível em: <https://mail.mozilla.org/pipermail/es-discuss/2008-August/003400.html> 10 de março de 2019

[17] CECHINEL, Alexandre et al. Avaliação do framework Angular e das bibliotecas React e Knockout para o desenvolvimento do Frontend de aplicações Web. 2017.

[18] Reconciliation - react. Disponível em: <https://reactjs.org/docs/reconciliation.html> Acesso em: 20/04/2019

[19] javascript - can anyone explain the difference between reacts one-way data binding and angular's two-way data binding - stack overflow. Disponível em: https://stackoverflow.com/a/37566693. Acesso em: 9 de março de 2019

[20] CECHINEL, Alexandre et al. Avaliação do framework Angular e das bibliotecas React e Knockout para o desenvolvimento do Frontend de aplicações Web. 2017.

[21] Knockout. Disponível em: <https://knockoutjs.com/documentation/ introduction.html>. Acesso em: 02 de abr. de 2019

[22] Flux | application architecture for building user interfaces. Disponível em: https://facebook.github.io/flux/docs/in-depth-overview.html#content Acesso em: 30 abr. 2019

[23] Read me redux. Disponível em: https://redux.js.org . Acesso em: 9 de março de 2019

[24] JADHAV, Madhuri A.; SAWANT, Balkrishna R.; DESHMUKH, Anushree. Single page application using angularjs. International Journal of Computer Science and Information Technologies, v. 6, n. 3, p. 2876-2879, 2015.

[25] SILBERSCHATZ, ABRAHAM, KORTH, HENRY FSUDARSHAN, S. Database systems concepts. Estados Unidos: McGraw-Hill Companies, Inc., 2011.

[26] AVEDA, SCOTT. What is the importance of a database management system. www.linkedin.com. Disponível em: <https://www.linkedin.com/pulse/what-importancedatabase-management-system-scott-aveda/ >. Acesso em: 9 de março de 2019

[27] ANTUNES, Igor Leal. Avaliação e Evolução Arquitetural de um Sistema Web ERP. 2017. 92 f. TCC (Graduação) - Curso de Ciência da Computação, Centro de Informática, Universidade Federal da Paraíba, João Pessoa, 2017.

[28] SULLIVAN, DAN. NoSQL for mere mortals. Boston: Addison-Wesley, 2015.

[29] REDIS. 2015. Disponível em: <https://redis.io/>. Acesso em: 20 dez. 2018.

[30] MONGODB. 2015. Disponível em: <https://www.mongodb.com/>. Acesso em: Acesso em: 20 dez. 2018.

[31] THE APACHE SOFTWARE FOUNDATION. **Cassandra.** 2016. Disponível em: <http://cassandra.apache.org/>. Acesso em: 20 dez. 2018.

[32] Neo4j. **Neo4j.** Disponível em: <https://neo4j.com/>. Acesso em: 20 dez. 2018.

[33] DE SOUZA, ARICÉLIOCÂNDIDO, PETRÔNIO. Comparativo técnico de tecnologias de banco de dados: Relacional, NoSQL e NewSQL. Disponível em: <https://www.academia.edu/6559619/Comparativo_T%C3%A9cnico_entre_tecnologias_de_banco_de_dados_relacional_NoSQL_e_NewSQL>. Acesso em: 1 de março de 2019

[34] ENTENDENDO o Pattern Model View ViewModel MVVM. Disponível em: <https://www.devmedia.com.br/entendendo-o-pattern-model-view-viewmodel-mvvm/18411>. Acesso em: 3 jan. 2019

[35] JQUERY.ORG, JQUERY. jQuery. Jquery.com. Disponível em: <https://jquery.com/>. Acesso em: 2 de fevereiro de 2019.

[36] JQUERY.ORG, JQUERY. jQuery UI. Jqueryui.com. Disponível em: <https://jqueryui.com/>. Acesso em: 2 de fevereiro de 2019.

[37] OTTO, MARK. Bootstrap. Getbootstrap.com. Disponível em: <http://getbootstrap.com/>. Acesso em: 2 de fevereiro de 2019.

[38] Taggd - Add notes to your images, responsively. Timseverien.com. Disponível em: <https://timseverien.com/taggd/v3/>. Acesso em: 2 de fevereiro de 2019.

[39] Introduction to Raphaël.js - HTML5 Rocks. HTML5 Rocks - A resource for open web HTML5 developers. Disponível em: <https://www.html5rocks.com/en/tutorials/raphael/intro/>. Acesso em: 2 de fevereiro de 2019.

[40] FPDF. Fpdf.org. Disponível em: <http://www.fpdf.org/>. Acesso em: 2 de fevereiro de 2019.

[41] FFmpeg. Ffmpeg.org. Disponível em: <https://www.ffmpeg.org/>. Acesso em: 2 de fevereiro de 2019

[42] - MAGALHÃES, Ícaro Lima. Um estudo comparativo entre padrões arquiteturais para o desenvolvimento de aplicativos para a plataforma iOS. 2018. 51 f. TCC (Graduação) - Curso de Ciência da Computação, Centro de Informática, Universidade Federal da Paraíba, João Pessoa, 2018.

[43] PANO, Amantia; GRAZIOTIN, Daniel; ABRAHAMSSON, Pekka. What leads developers towards the choice of a JavaScript framework?, 2016.

[44]PADRÃO de Projeto Observer em Java. Disponível em: <https://www.devmedia.com.br/padrao-de-projeto-observer-em-java/26163>. Acesso em: 1 maio 2019

[45] APPLICATION Program Interface. Disponível em: <http://foldoc.org/Application+Program+Interface>. Acesso em: 1 maio 2019

[46] BOSCARIOLI, Clodis et al. Uma reflexão sobre banco de dados orientados a objetos. Congresso de Tecnologias para Gestão de Dados e Metadados do Cone Sul, Paraná, Brasil. 2006

APÊNDICE A – DOCUMENTO DE REQUISITOS DO SISTEMA POS/ERP DA EMPRESA X

Igor Leal Antunes, Luiz Henrique Freire Barros

Índice

1. Introdução

1.1. Propósito do documento

Esse documento tem por objetivo apresentar os requisitos que os sistemas *POS/ERP* da Empresa X deve atender em diferentes níveis de detalhamento. Nele está contido informações gerais referente ao sistema e à empresa.

1.2. Visão geral do documento

Esta introdução fornece as informações necessárias para fazer um bom uso deste documento, explicitando seus objetivos e as convenções que foram adotadas no texto, além de conter uma lista de referências para outros documentos relacionados. As demais seções apresentam a especificação dos sistemas POS/ERP e estão organizadas como descrito abaixo.

- Seção 2 – Descrição geral do sistema: apresenta uma visão geral do sistema, caracterizando qual é o seu escopo e descrevendo seus usuários.
- Seção 3 – Glossário: Apresenta uma definição de todos os atributos e domínios das entidades.
- Seção 4 – Análise: específica todos os requisitos funcionais e não funcionais do sistema.
- Seção 5 – Especificação de requisitos: esta seção fornece uma explanação sobre os casos de uso da aplicação, os atores.
- Seção 6 - Descrição de interface com o usuário: nesta seção é mostrado modelos e desenhos de telas do sistema que são necessárias para esclarecer alguns requisitos do mesmo.

1.3. Documentos Relacionados

1. BARROS, Luiz. Proposta de Reengenharia Para um Sistema Web POS/ERP. 2019. TCC (Graduação). Curso de Ciência da Computação, Centro de Informática, Universidade Federal da Paraíba, João Pessoa
2. ANTUNES, Igor Leal. Avaliação e Evolução Arquitetural de um Sistema Web ERP. 2017. 92 f. TCC (Graduação) - Curso de Ciência da Computação, Centro de Informática, Universidade Federal da Paraíba, João Pessoa, 2017

2. Descrição Geral

Nos dias atuais, qualquer organização empresarial precisa de algum tipo de Sistema de informação para auxiliar suas atividades diárias. Muitas vezes, essas atividades demandam de um complexo gerenciamento de informação, tornando sistemas POS/ERP indispensáveis para essas empresas.

Dentro deste contexto, tem-se o sistema da empresa X. Ele foi desenvolvido para trabalhar no ramo de confecção de roupas de luxo sob medida. O sistema oferece serviços dentro de setores como vendas, controle de qualidade, produção, acabamento, logística e etc. Cada setor possui processos bem definidos que devem funcionar por meio do sistema citado. Basicamente os diferentes pontos de venda, distribuídos geograficamente, coletam informações para que a roupa seja confeccionada, tais como: medidas corporais dos clientes, design escolhido de cada roupa, tipo de corpo e possíveis imperfeições de cada cliente, entre outros dados.

A peça pode ser fabricada em diferentes regiões do mundo dependendo de vários fatores, como a disponibilidade e custos de material bruto para confecção e de mão de obra. E o sistema deve apoiar o controle dessa confecção. Depois que a peça está pronta, ela sofre um processo de controle de qualidade, onde ela é enviada para um dos pontos de venda escolhidos pelo cliente, onde ele pode provar a roupa. Se algo não estiver correto, ela passa por ajustes. Esse processo se repete até o cliente estar satisfeito. Com isso, é necessário enviar os produtos para diferentes localizações, e para isso, o sistema precisa dar apoio a toda essa logística.

O sistema é dividido em duas partes fundamentais, o POS e o ERP (lado do cliente e do servidor respectivamente) e eles controlam todos os processos da empresa, desde a venda até a entrega dos produtos, passando por um processo de validação como mencionado anteriormente.

POS vem do inglês point of sale, e é a parte do sistema com a qual o vendedor interage com o cliente, seja para coletar informações de pagamento, cadastrar um cliente, ou para auxiliar o vendedor a personalizar a peça que o cliente está comprando.

ERP vem do inglês *Enterprise Resource Planning,* e é a parte do sistema onde é feita a análise de todos os dados coletados pela empresa, dado esse voltado para lidar com as atividades rotineiras da empresa.

O sistema também é responsável pela análise dos dados da companhia como a quantidade de peças que um determinado funcionário vendeu; o lucro total em um ano; qual peça, e design são mais lucrativos; logística; perfil dos usuários, entre outros fatores.

O sistema está em operação desde o ano de 2014, e toda documentação aqui apresentada foi extraída a partir da análise do sistema já em funcionamento e seu respectivo código.

2.1. Suposições e Restrições Gerais

Para as funcionalidades de todos os usuários é necessário que estes possuam conhecimento dos serviços disponibilizados pelo sistema e como utilizá-los de forma correta.

Por questão de segurança, os dados dos usuários devem ser mantidos encriptografados. O sistema só executaria as operações depois que o usuário realizasse o login no sistema, inserindo seu nome de usuário e senha.

Existem diferenças entre as seções dependendo do tipo de usuário logado.

2.2. Usuários

Com base no entendimento do negócio da empresa através da análise de código do sistema da empresa X, é possível identificar quatro tipos de perfis de usuário, são eles: vendedor, alfaiate, administrador e cliente.

2.2.1. Vendedor

Usuário do POS, ele é responsável por atender o cliente em um dos pontos de vendas e realizar o processo de venda, ou de "*fitting*" (auxiliar o cliente a provar a peça e se ela atende aos requisitos do cliente)

2.2.2. Alfaiate

Ele baixa um PDF do ERP e a partir dele, o alfaiate monta a peça de acordo com as requisições do cliente. Se o cliente possuir alguma queixa durante o *fitting*, a peça volta para a fábrica e o alfaiate faz os ajustes necessários

2.2.3. Administrador

Possui acesso total ao sistema, tanto no POS como no ERP. Ele pode acompanhar os pedidos e realizar um novo. Ele é o único que pode aprovar um pedido depois que ele é feito.

2.2.4. Cliente

Não atua diretamente em nenhum dos sistemas, exceto no ato do pagamento, onde ele insere informações como método de pagamento e sua assinatura.

3. Glossário

No domínio da alfaiataria, o conhecimento de alguns termos são necessários para entender a realização do negócio, como por exemplo, medidas corporais, padrões corporais, blazer, dentre outros. A seguir alguns termos são explicados.

3.1. Medidas corporais

São informações, em centímetros, de quanto mede uma parte do corpo.

As partes medidas são:

- Peito
- Estômago
- Ancas
- Ombro
- Manga esquerda
- Manga direita
- Frente
- De volta
- Bicep
- Braços dianteiros
- Pulso
- Comprimento de lapela
- Pescoço
- Zero pontos
- Frente Esquerda para Zero
- Costas esquerda para zero
- Frente direita para zero
- Costas direita para zero
- Bicep Tenso
- Pantufas
- Ancas
- Coxa
- Algemas
- frente da virilha
- Inseam
- Virilha
- Zíper
- Joelho

3.2. Padrões corporais

Padrões corporais está relacionado com a postura do cliente. A **tabela 01**, mostra quais os tipos de postura que a empresa considera para cada parte do corpo e a **Figura 01** mostra um exemplo de padrão corporal para ombros.

Parte do Corpo	Opções (Língua Original)
Ombros	Both slightly sloped
	Both very sloped
	Normal
	Left normal Right sloped
	Left slightly sloped Right very sloped
	Right normal Left sloped
	Right slightly sloped Left very sloped
	Square
Braços	Backwards arms
	Balanced
	Forwards arms
Trás	Curved back
	Hump back
	Normal
	Upper back curve
	Upper back curved forward
Frente	Caving in
	Normal
	Slightly protruding
	Chicken

	Well padded chest
Pernas	Bow legged
	Inwards
	Normal
	Outwards
Pescoço	Long
	Normal
	Very Short
Postura	backward
	Neutral
	Slight lean
	Strong lean
Barriga	Beer belly
	Bulge
	Normal
	Pot belly
	Slight bulge
	Washboard
Tornozelo	Heavy legs
	Large muscular thighs
	Thick thighs
	Thin legs
	Very erect prominent calves
	Normal

Tabela 01 - Padrões corporais para cada parte do corpo

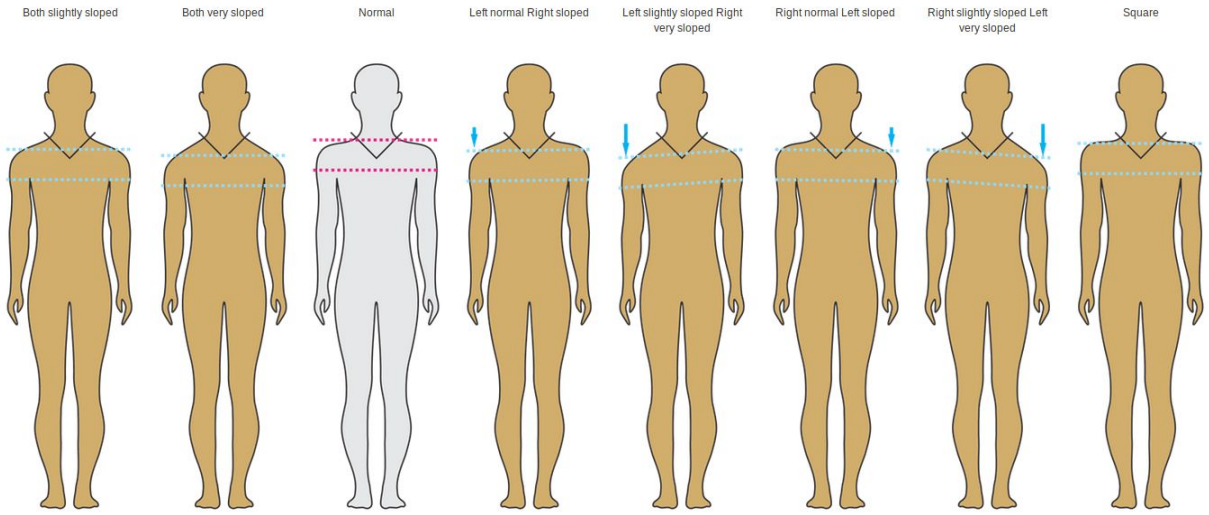


Figura 01 - Exemplo de padrão corporal para ombros.

Imagem extraída da aplicação.

3.3. Peça de roupa

A empresa X trabalha atualmente com 5 tipos de roupas: blazers, calças, ternos, coletes, e camisas. Sendo um terno o conjunto formado por um blazer e uma calça. A **Figura 02** a seguir ilustra cada uma das peças.

Figura 02 - Da esquerda para direita tem-se uma representação de um blazer, uma calça, um colete, e uma camisa.

Fontes: Terno [1], calça [2], colete [3], camisa [4]

3.4. Design, Categoria e Opções

Especificações de uma peça. Cada uma das peças que a empresa X vende, pode ser personalizada de acordo com as requisições do cliente. Por exemplo, a lapela de um blazer pode ter os designs ilustrados na **Figura 03**.

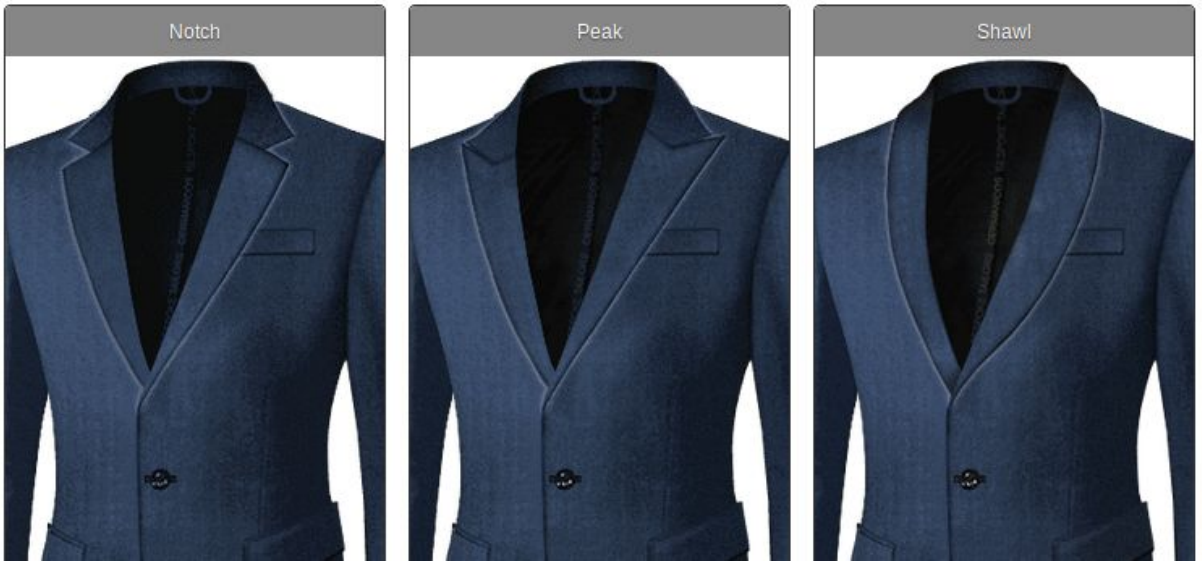


Figura 03 - Exemplo de personalização de um blazer

Diversas partes de cada terno (categorias) possuem diversos tipos de designs diferentes (opções). No exemplo acima, A “lapela” é a categoria e “Notch”, “Peak”, “Shawl” são as opções correspondentes para esta categoria.

Na **Figura 04**, é possível ver outro exemplo de design. Desta vez, é o tipo de manga de uma camisa pode ser personalizado de acordo com a preferência do cliente.

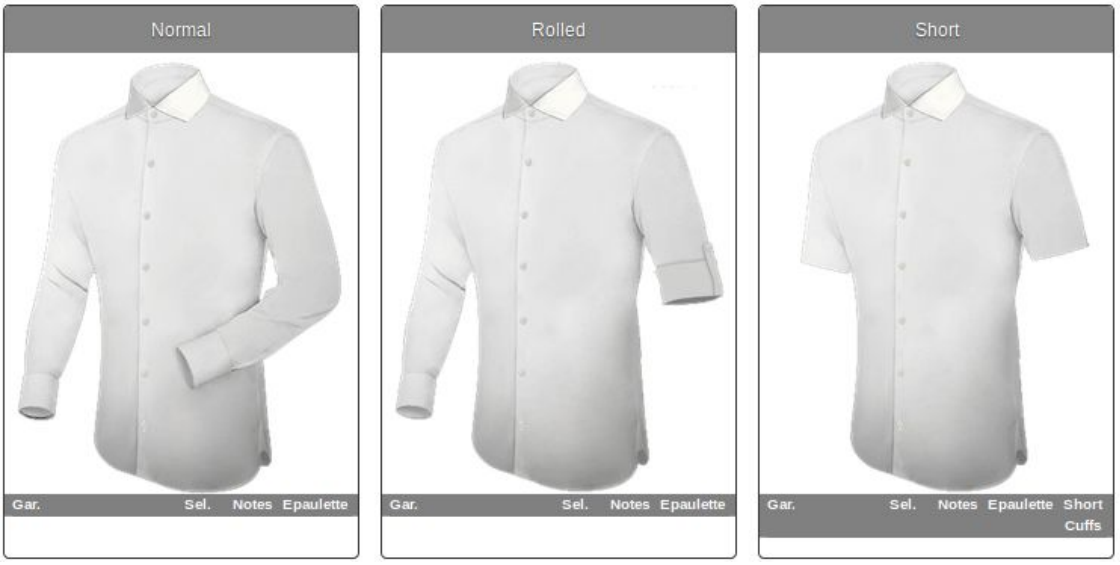


FIgura 04 - Exemplo de personalização da manga de uma camisa

3.5. Pedido

Um pedido é a ordem de compra; encomenda.

Um exemplo de pedido, seria:

O cliente Fulano encomendou uma calça com as seguintes especificações:

- *Tecido: FLA 11*
- *Ajuste: FIT*
- *Estilo: Narrow Slim*
- *Pregas: Triple Pleats*
- *Bolso da frente: Slanted Welt*
- *Bolso de trás: sem bolso atrás*
- *Cinto: Single Straight*
- *Comentários Gerais: "Perna esquerda 1 polegada maior que a direita"*

3.6. Fitting

Nome chamado ao processo de teste da roupa. Depois que um pedido é submetido e a peça é concluída, a roupa é enviada para um dos diversos pontos de venda escolhidos pelo cliente e lá ele pode provar a roupa e dizer se todos os requisitos do design foram atendidos.

3.7. PDF

Um documento no formato PDF que pode ser impresso pelos funcionários na fábrica. Com este documento, os alfaiates podem confeccionar a peça de acordo com a opções de design escolhidas pelos clientes.

Este documento possui todas as informações relevantes para a fabricação tais como medida corporal do cliente (seção 3.1), padrão corporal (seção 3.2) e design escolhido pelo cliente (seção 3.4).

4. Requisitos do Produto

A partir da análise do sistema da empresa X, também conseguimos extrair parte dos requisitos funcionais e não funcionais do produto.

4.1. Requisitos Funcionais

[RF01] Registrar um pedido

Descrição: O sistema deve ser capaz de registrar um pedido realizado pelo usuário no perfil de vendedor. Um pedido é caracterizado pelas seguintes propriedades: tipo de peça, medidas corporais, padrão corporal, e por fim, fotos e vídeos do cliente vestindo uma roupa auxiliar[1].

[RF02] Atualizar medidas dos clientes

Descrição: O sistema permitir ao administrador que atualize as medidas corporais de cada cliente.

[RF03] Atualizar design do produto

Descrição: O sistema permite que o administrador altere o design de cada roupa.

[RF04] Atualizar o padrão corporal de cada cliente

Descrição: O sistema permite que o administrador atualize o padrão corporal de cada cliente.

[RF05] Gerar arquivo PDF para produção

Descrição: O sistema permite aos usuários administradores e de controle de produção que seja gerado um arquivo no formato PDF com todos os detalhes necessários para que sejam produzidos os produtos.

[1] Roupa contendo marcações especiais que ajudam o alfaiate a fabricar a roupa.

[RF06] Organizar a logística

Descrição: O sistema permite aos usuários administradores e de controle de produção, que agrupem os produtos e definem a data e o destino de cada grupo, para que os produtos possam ser enviados para outros destinos de uma única vez.

[RF07] Agendar Provas

Descrição: O sistema permite aos administradores e vendedores que sejam agendados encontros com os clientes, para que se defina se a roupa serve perfeitamente ou se são necessárias correções na roupa.

[RF08] Processo de correção

Descrição: O sistema permite que os administradores encaminhem produtos para que sejam feitas correções, ou seja, permite que os administradores marquem o produto para ser enviado para outros locais. As correções podem ocorrer em diferentes cidades com diferentes alfaiates.

[RF09] Receber pagamentos

Descrição: O sistema permite que usuários vendedores e administradores recebam pagamentos dos clientes. Os pagamentos podem ser feitos pelo POS e enviados para o ERP via endpoint, ou feito diretamente no sistema ERP.

[RF10] Finalizar pedido

Descrição: O sistema permite que os administradores concluam um pedido, uma vez que todos os processos tenham sido concluídos com êxito.

[RF11] Tradução de PDFs

Descrição: O sistema permite que os administradores possam traduzir comentários e instruções escritas em inglês para a língua nativa do país onde a roupa será fabricada.

[RF12] Atualização da forma corporal de cada cliente

Descrição: Após o fim de cada pedido, o sistema permite que os administradores possam atualizar a forma corporal (localizada fisicamente na fábrica em que a roupa foi produzida) de cada cliente.

[RF13] Notificação de erros de produção

Descrição: O sistema permite que administradores notifiquem a sede da empresa de qualquer tipo de problema ou dúvidas durante a produção do produto.

[RF14] Criar novos clientes

Descrição: O sistema permite a criar novos clientes

[RF15] Realizar login

Descrição: O sistema só pode ser utilizado mediante login informando nome de usuário e senha.

[RNF12] Backup de Dados

Descrição: O sistema fará um backup parcial do seu banco de dados automaticamente

4.2. Requisitos não funcionais

[RNF01] Especificação de Software

Descrição: O sistema ERP deve ser desenvolvido em linguagem de programação PHP versão 5.6.10 enquanto o POS deve ser desenvolvido em Javascript, com HTML5 e CSS3

[RNF02] Banco de Dados

Descrição: O sistema faz uso do banco de dados relacional MySQL versão 5.5.57-cll para Linux RedHat, para manter os dados do sistema.

[RNF03] Servidor

Descrição: O sistema deve funcionar no servidor Apache/2.2.31 utilizando FastCGI e compressão Gzip, além de suportar os protocolo HTTP/1.1

[RNF04] Versão da biblioteca Jquery UI

Descrição: O sistema deve utilizar a versão 1.11.4 da biblioteca Jquery UI

[RNF05] Versão da biblioteca Bootstrap

Descrição: O sistema deve utilizar a versão 2.1.1 da biblioteca Bootstrap

[RNF06] Versão da biblioteca Raphael.js

Descrição: O sistema deve utilizar a versão 2.0.1 da biblioteca Raphael.js

[RNF07] Versão da biblioteca Tagdd.js

Descrição: O sistema deve utilizar a versão 2.0da biblioteca Tagdd.js

[RNF08] Versão da biblioteca FFPDF

Descrição: O sistema deve utilizar a versão 1.6 da biblioteca FFPDF

[RNF08] Versão da biblioteca FFMPEG

Descrição: O sistema deve utilizar a versão 0.10.16 da biblioteca FFMPEG

[RNF09] Versão da biblioteca GroceryCrud

Descrição: O sistema deve utilizar a versão 1.5.4 da biblioteca GroceryCrud

[RNF10] Versão da biblioteca Knockout.js

Descrição: O sistema deve utilizar a versão 3.4.2 da biblioteca Knockout.js

[RNF10] Recebimento de dados via End-point

Descrição: Todas as transações de dados feitas com o sistema POS devem passar por autenticação primeiro. A autenticação deve verificar nome de usuário e senha. Os dados de senha devem vir do POS através de uma requisição HTTPS utilizando certificados TSL/SSL.

[RNF11] Comunicação segura entre cliente e servidor

Descrição: O sistema se comunica através de protocolo HTTPS e utilizar certificados no lado do servidor do tipo SSL e TSL, mais especificamente o OpenSSL. Implementando os algoritmos PKCS SHA-256 com criptografia RSA

[RNF12] Periodicidade dos Backups de Dados

Descrição: O sistema fará um backup parcial do seu banco de dados todos os dias e guardará os dados por 1 mês.

[RNF13] Dados sensíveis

Descrição: Todos os dados sensíveis do sistema tais como: senhas ou dados importantes de clientes, como cartões de crédito, devem ser armazenados de forma criptografada com o algoritmo BCrypt.

[RNF14] Comunicação entre sistemas

Descrição: Os sistemas ERP e POS devem se comunicar utilizando o protocolo HTTP POST. Respostas a cada requisição devem ser informadas.

[RNF15] Dispositivos suportados

Descrição: O ERP deve ser suportado por dispositivos Desktop com acesso à Internet, através dos browsers Mozilla Firefox 20+ com JavaScript ativado. Já o POS, deve ser suportado por tablets iPad

[RNF16] Idioma

Descrição: O sistema deve apresentar a interface com o usuário na língua inglesa.

[RNF17] Idioma dos PDFs

Descrição: Os PDFs gerados devem estar disponíveis em inglês e também na língua nativa do país de produção do produto.

[RNF18] Informar Erro de Comunicação

Descrição: Em caso de erro na comunicação entre os softwares POS e ERP, o sistema deve indicar o erro na resposta.

[RNF19] Divisão do sistema

Descrição: O sistema deve ser dividido em dois, um POS e um o ERP.

5. Casos de Uso

5.1. Descrição dos Casos de Uso

[UC01] - Submeter Vendas

Descrição	Processo inicial de venda e aquisição de informações dos clientes. A produção dos produtos será baseada nesses dados.
Ator	Vendedor e Cliente. O cliente informa ao vendedor como ele quer o design, mas só o vendedor atua diretamente no POS. O cliente atua diretamente apenas durante a etapa de pagamento.
Pré-condições	1. Existir Cliente cadastrado no sistema
Pós-condições	Pedido estará submetido e estará disponível para acesso e modificações no sistema ERP por administradores.
Fluxo Principal	1. Realizar a captura de informações do pedido 2. Vendedor submete pedido para o ERP
Fluxo Secundário	**- Cliente não existente** Caso o cliente não esteja cadastrado ainda, o usuário deve submeter as informações do novo cliente primeiro.

[UC02] – Processar Pagamentos

Descrição	Clientes poderão efetuar pagamentos referentes aos seus pedidos
Ator	Vendedor, Administrador e Cliente
Pré-condições	Necessário que o cliente já possua ordem não finalizada dentro do sistema
Pós-condições	Saldo devedor do cliente será reduzido ou zerado.
Fluxo Principal	1. Clientes submetem pagamentos pelo sistema POS juntamente com os vendedores 2. Pagamento é enviado ao ERP através de requisições ajax
Fluxo Secundário	**- Vendedor não presente** Clientes submetem pagamentos pelo sistema ERP juntamente com os administradores.

[UC03] – Agendar de provas

Descrição	Vendedores e Administradores agendam sessão de *fitting* para que os clientes possam provar os produtos já manufaturados. O processo de prova é necessário para que se identifique possíveis erros ou imperfeições no produto.
Ator	Vendedor, Administrador e Cliente
Pré-condições	Necessário que os produtos dos clientes já estejam manufaturados e prontos para serem provados
Pós-condições	Ordem é finalizada ou produto é redirecionado para alterações de correção
Fluxo Principal	1. Vendedores ou administradores entram em contato com os clientes e marcam uma sessão no dia e hora convenientes para os dois. 2. Clientes provam e avaliam o produto final juntamente

	com o vendedor 2.1. [Extends [UC11] – Processo de correção de roupas] 2.2. [[Extends [UC04] – Atualizar medidas] 2.3. [[Extends [UC05] – Atualizar design] 2.4. [[Extends [UC05] – Atualizar o padrão corporal] 3. Ordem finalizada
Fluxo Secundário	-

[UC04] – Atualizar medidas

Descrição	Administradores podem atualizar valores de medidas corporais dos clientes para que os dados gerados de produção sejam atualizados.
Ator	Administradores
Pré-condições	Necessário que o cliente já possua cadastro de medidas no sistema
Pós-condições	Próximos PDFs de produção serão gerados com os valores mais atuais de medidas. E os valores antigos deverão ir para o histórico de medidas dos clientes
Fluxo Principal	1. No menu de dados de cada cliente, administradores deverão inserir novas medidas
Fluxo Secundário	-

[UC05] – Atualizar Design

Descrição	Administradores podem atualizar o design de cada produto dos clientes, para que os produtos possam ser produzidos corretamente.
Ator	Administradores
Pré-condições	Necessário que o cliente já possua pelo menos uma ordem não finalizada e não em produção

Pós-condições	Próximos PDFs de produção serão gerados com os valores mais atuais de design. E os valores antigos deverão ir para o histórico de design dos clientes
Fluxo Principal	1. No menu de dados de cada ordem, administradores devem escolher o produto desejado 2. Depois devem clicar em "Design" e serão levados a página de design do produto. 3. Após feitas as alterações os administradores deverão salvar os dados
Fluxo Secundário	-

[UC06] – Atualizar padrão corporal

Descrição	Administradores podem atualizar tipo de de formato corporal de cada cliente, com isso, os PDFs de produção terão informações mais precisas para a produção dos produtos
Ator	Administradores
Pré-condições	Necessário que o cliente já possua cadastro de formato corporal no sistema
Pós-condições	Próximos PDFs de produção serão gerados com os valores mais atuais de formato corporal. E os valores antigos deverão ir para o histórico de formatos corporais
Fluxo Principal	1. Dentro dos clientes, os administradores deverão clicar na opção de editar formato corporal 2. Após efetuar as mudanças devem clicar em Salvar
Fluxo Secundário	-

[UC07] – Organizar logística

Descrição	Administradores e fabricantes podem organizar como deve ser feito os processos de logística de cada produto, quando é necessário que certos processos (produção, ajustes) sejam efetuados em outros locais.

Ator	Administradores, fabricantes e alfaiates
Pré-condições	Necessário que exista um produto a ser entregue em outra localização
Pós-condições	Metadados de localização de cada produto serão atualizados dentro do sistema.
Fluxo Principal	1. No menu de administradores, o usuário deve clicar em *Manage shipping boxes* (do inglês, Organizar logística de produtos) 2. Os usuários devem arrastar cada produto para suas determinadas caixas, caso não exista uma caixa adequada (com data de envio e destino requerido), o usuário pode criar novas caixas. 3. Após enviar fisicamente a caixa, o usuário deve indicar isso no sistema e preencher mais dados, tais como o custo de envio e código de rastreamento
Fluxo Secundário	1. No menu de fabricantes, o usuário deve clicar em *Manage shipping boxes* (do inglês, Organizar logística de produtos) 2. Os usuários devem arrastar cada produto para suas determinadas caixas, caso não exista uma caixa adequada (com data de envio e destino requerido), o usuário pode criar novas caixas. 3. Após enviar fisicamente a caixa, o usuário deve indicar isso no sistema e preencher mais dados, tais como: custo de envio e código de rastreamento

[UC08] – Geração de PDF para produção

Descrição	Administradores e fabricantes podem gerar PDF para produção. Esse PDF conterá todas as informações necessárias para que os produtos sejam manufaturados
Ator	Administradores e fabricantes
Pré-condições	Necessário que o cliente já possua uma ordem que ainda não está em produção
Pós-condições	Após gerado o PDF e a ordem for aprovada para produção, os fabricantes poderão encaminhar os PDFs para que os produtos sejam manufaturados.

Fluxo Principal	1. Dentro dos detalhes de cada produto, o Administrador deve aprovar todos os dados – medidas, design, formato corporal e data de entrega e então o pdf será gerado em inglês. 2. Fabricante deve verificar se existe algo a ser traduzido para a língua nativa da fábrica. 3. 3.1. [Extends [UC09] – Tradução de PDF de produção] 4. Depois os fabricantes terão que gerar o PDF na língua nativa e só então, efetuar a produção do produto.
Fluxo Secundário	**- Impossibilidade de produção** Caso exista alguma informação não clara ou errada, o fabricante deverá notificar os administradores através da criação de erros dentro do sistema e só depois a produção deverá continuar normalmente

[UC09] – Tradução de PDF de produção

Descrição	Fabricantes devem traduzir informações essenciais para a produção do produto na língua nativa da fábrica, uma vez que um PDF na língua nativa dos fabricantes resulta em um produto final com mais qualidade.
Ator	Fabricantes
Pré-condições	Necessário que o PDF em inglês já tenha sido aprovado pelos administradores
Pós-condições	Um PDF na língua solicitada, pronto para ser utilizado na produção do produto deve ser gerado
Fluxo Principal	1. Dentro dos detalhes de cada produto, os administradores devem traduzir todos os campos essenciais para a produção. Os campos serão indicados pelo sistema e será fornecida uma interface de tradução adequada para cada campo automaticamente. 2. Após todos os campos forem traduzidos, a geração de PDF será liberada na língua desejada e a produção poderá começar.
Fluxo Secundário	-

[UC10] – Notificar erros de produção

Descrição	Fabricantes podem notificar administradores de erros ou dúvidas durante os processos de produção.
Ator	Fabricantes e Administradores
Pré-condições	Necessário que o produto já esteja em processo de produção
Pós-condições	Tickets serão criados no sistema para manter histórico de erros.
Fluxo Principal	1. Durante a produção, os fabricantes podem notar alguma inconsistência nos dados – Medidas dos clientes, design da roupa, formato corporal e outros. 2. Caso algo impeça a produção do produto, ele deve criar um ticket dentro da janela de dados sobre a ordem. 3. Administradores devem ser notificados e respondem imediatamente para que a produção possa ser continuada. Em casos mais graves, o cliente deve ser contatado para esclarecer qualquer problema.
Fluxo Secundário	-

[UC11] – Processo de correção de roupas

Descrição	Após uma prova, podem ser encontrados problemas em cada produto, esses problemas devem ser corrigidos no processo de correção
Ator	Alfaiates, fabricantes, vendedores e administradores
Pré-condições	Necessário que existam produtos não perfeitos já completamente manufaturados
Pós-condições	O produto final terá correções efetuadas estará pronto para outro processo de prova.
Fluxo Principal	1. Caso o produto não sirva perfeitamente, vendedores ou administradores criarão um processo de alteração no menu principal do sistema. Deve ser informado qual produto será alterado, o que deve ser alterado, o

	alfaiate responsável pela alteração e a data limite. 2. Depois, deve-se encaminhar os produtos para o alfaiate designado, podendo ser necessário utilizar a seção de logística para isso. 2.1. [Extends [UC07] – Organização de logística] 3. Após o recebimento do produto, o alfaiate terá que confirmar os dados no sistema, efetuar as alterações e submeter um relatório do que foi feito. 4. Depois ele terá que enviar de volta o produto para a sua origem e um novo processo de prova será efetuado. 4.1. [Include [UC03] – Agendamento de prova] 5. Caso a alteração seja muito comprometedora, o formato corporal do cliente deve ser atualizado 5.1. [Extends [UC12] – Atualizar forma corporal]
Fluxo Secundário	-

[UC12] – Atualizar forma corporal

Descrição	Fabricantes podem atualizar fisicamente as formas corporais usadas na produção de cada produto. As formas são placas, que servem de molde para os produtos. Isso é muito importante para que se tenha um guia de como fabricar os produtos para cada cliente
Ator	Fabricantes
Pré-condições	Necessário que já exista informação de um produto que sirva bem em cada cliente.
Pós-condições	A forma corporal localizada na fábrica será alterada e os próximos produtos feitos para o cliente terão mais qualidade e precisão
Fluxo Principal	1. Após conseguir fabricar um produto que sirva bem aos clientes ou após alterações de correção do produto, os fabricantes devem ser notificados para que atualizem as formas corporais. 2. Na fábrica, os fabricantes irão alterar fisicamente a forma e indicar o dia, data e quais mudanças foram efetuadas.
Fluxo Secundário	-

[UC13] – Finalizar pedido

Descrição	Administradores devem finalizar pedidos. Isso significa que todas as pendências com o cliente estão concluídas.
Ator	Administradores
Pré-condições	Necessário que todas as condições sejam atendidas: Cliente esteja com os produtos, todos os pagamentos tenham sido efetuados e nenhuma pendência exista com os fabricantes e alfaiates.
Pós-condições	Informações sobre determinado pedido deixará de existir nos processos principais do sistema e será arquivado.
Fluxo Principal	1. Dentro da tela de informações de cada pedido os administradores deverão clicar no botão "Finalize Order". 2. O sistema então verificará se algumas pendências tenham sido alcançadas, tal como saldo devedor dos clientes e localização de cada produto. 3. Se tudo estiver nos conformes a ordem será arquivada, se não mensagens de erro serão lançadas no sistema.
Fluxo Secundário	-

[UC14] – Criar novos Clientes

Descrição	Na primeira venda, o cliente deve ser cadastrado para que a empresa possa manter contato.
Ator	Administradores e Vendedores
Pré-condições	-
Pós-condições	Informações sobre um cliente podem ser facilmente recuperadas para um pedido futuro.
Fluxo Principal	1. Da tela inicial, o ator entra na seção de usuários (CUSTOMERS) e preenche o formulário 2. Caso algum campo obrigatório esteja faltando, o

	sistema deve impedir a submissão e informar o dado faltando.
Fluxo Secundário	-

5.2. Diagrama

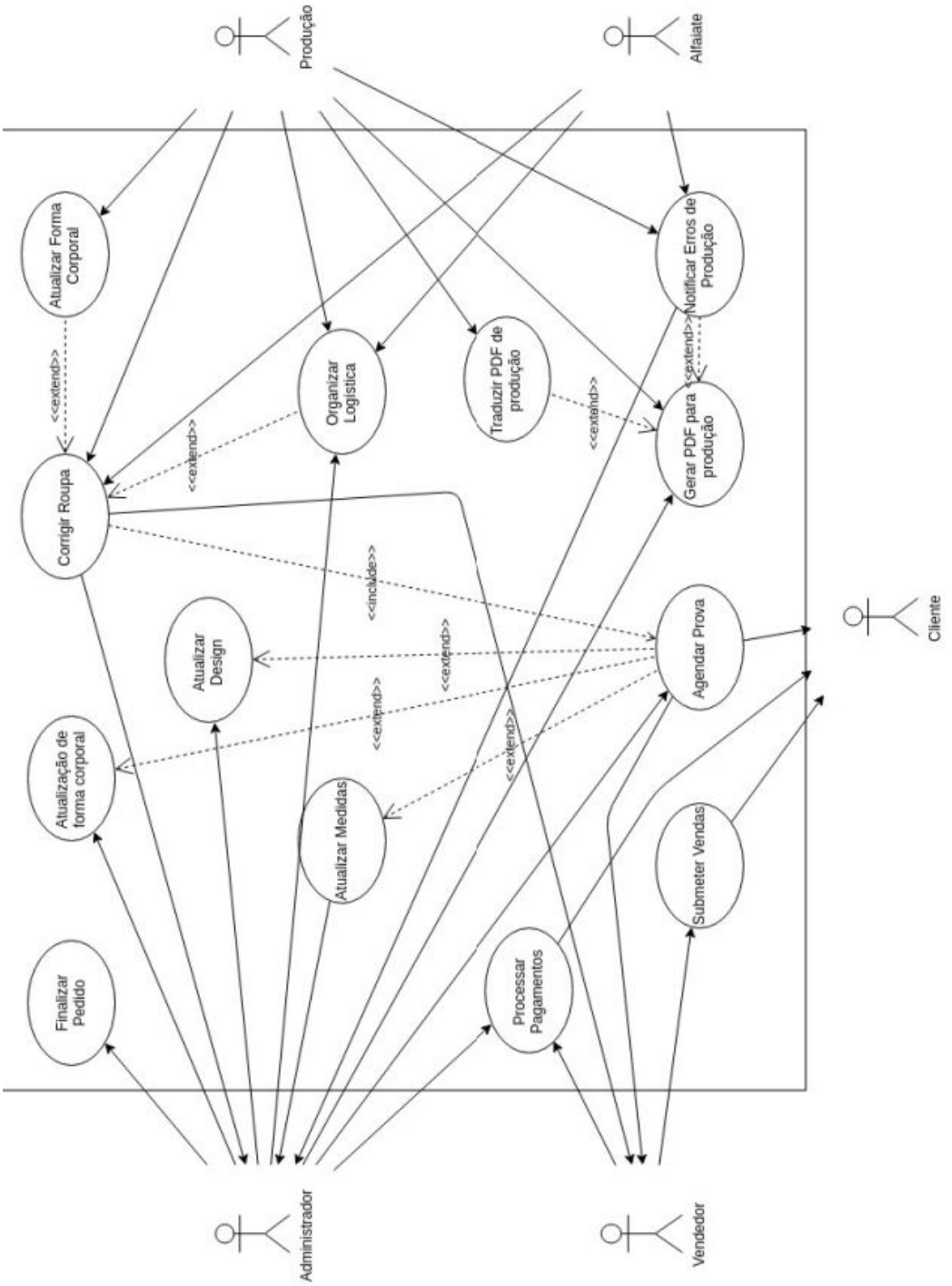
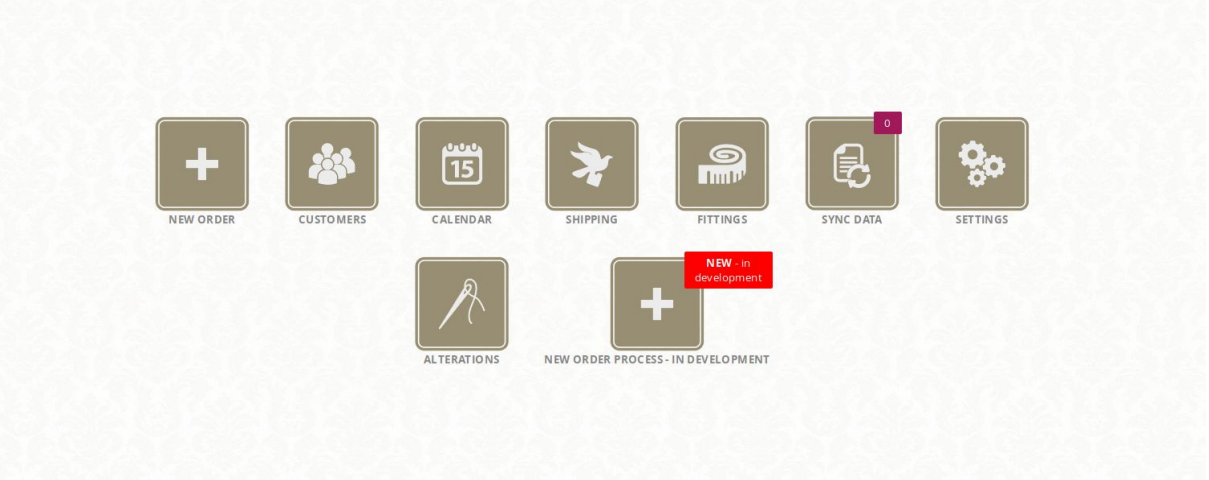


Figura 05 - Diagramas de caso de uso

6. Descrição de Interface[2]

6.1. Tela Inicial

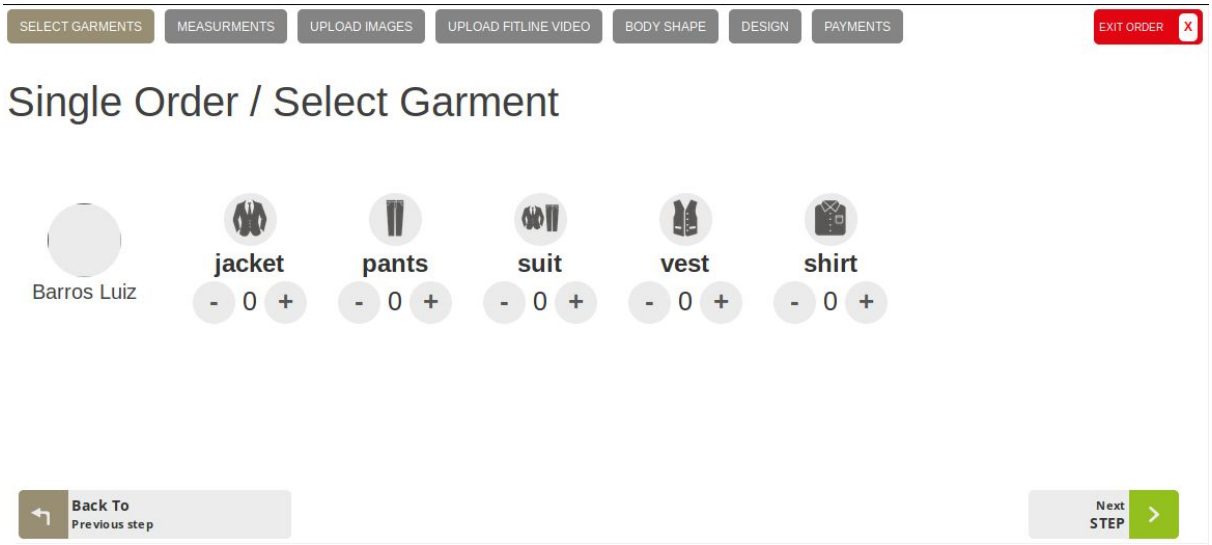


Esta é a tela inicial do POS. Dela é possível navegar entre as diferentes seções do sistema, como por exemplo:

- Criar um cliente (*CUSTOMERS)*;
- Ver informações de logística (SHIPPING);
- Iniciar o processo de fitting (*FITTING*);
- Iniciar um novo pedido (*NEW ORDER*);

[2] Por questões éticas, e de sigilo, só será possível mostrar algumas poucas telas do POS

6.2. Tela de Pedido 1

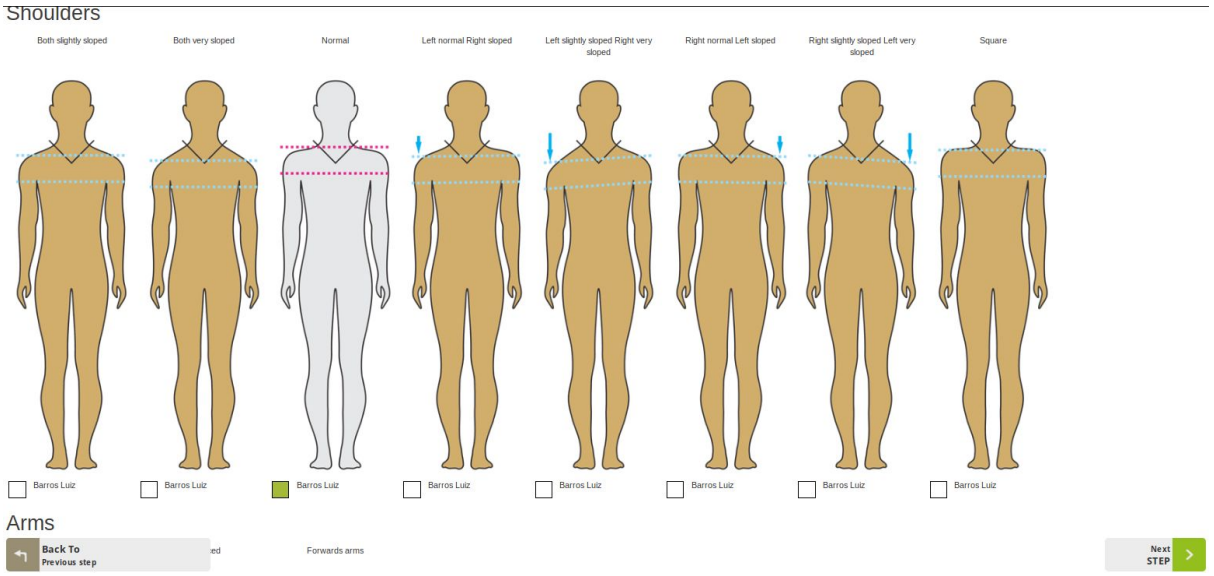


A realização do pedido é feito em várias etapas (podem ser vista na parte superior da tela) são elas:

- Seleção de peça de roupa (*SELECT GARMENT*)
- Realizar as medidas corporais (*MEASUREMENTS*)
- Tirar fotos do cliente vestindo uma roupa auxiliar (*UPLOAD IMAGES*)
- Filmar um vídeo do cliente usando essa roupa auxiliar (*UPLOAD FITLINES VIDEO*)
- Selecionar o padrões corporais (*BODY SHAPE*)
- Selecionar os designs (*DESIGNS*)
- E finalmente, a forma de pagamento (*PAYMENT*)

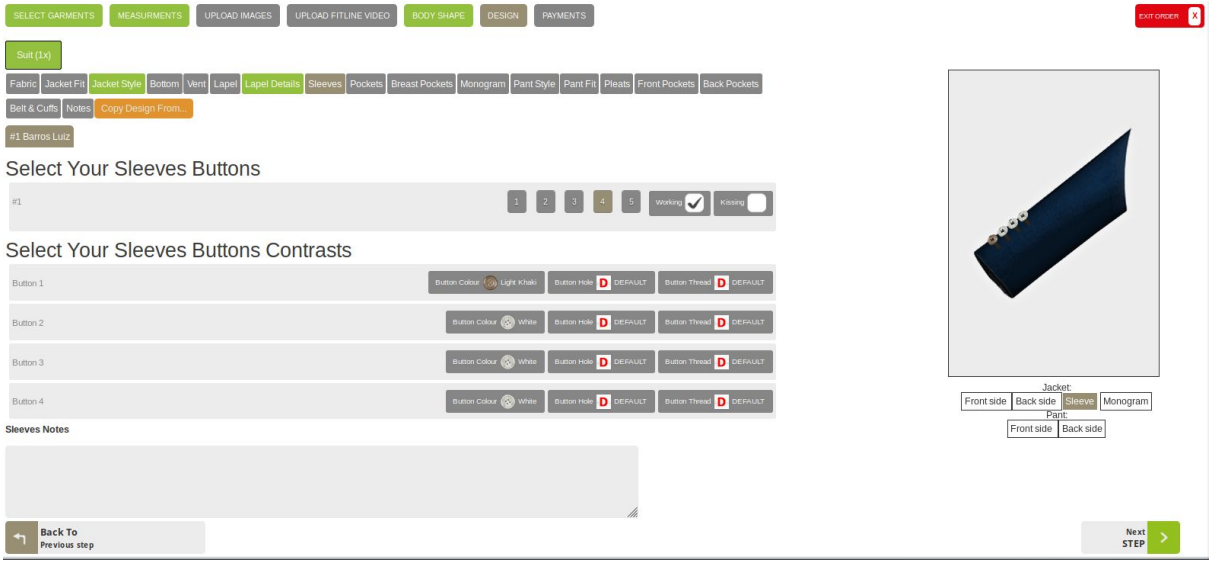
A etapa mostrada nesta seção é a de seleção de peça, pois, em um único pedido, é possível solicitar várias peças ao mesmo tempo, por exemplo, solicitar dois ternos e 2 camisas.

6.3. Tela de Pedido 2



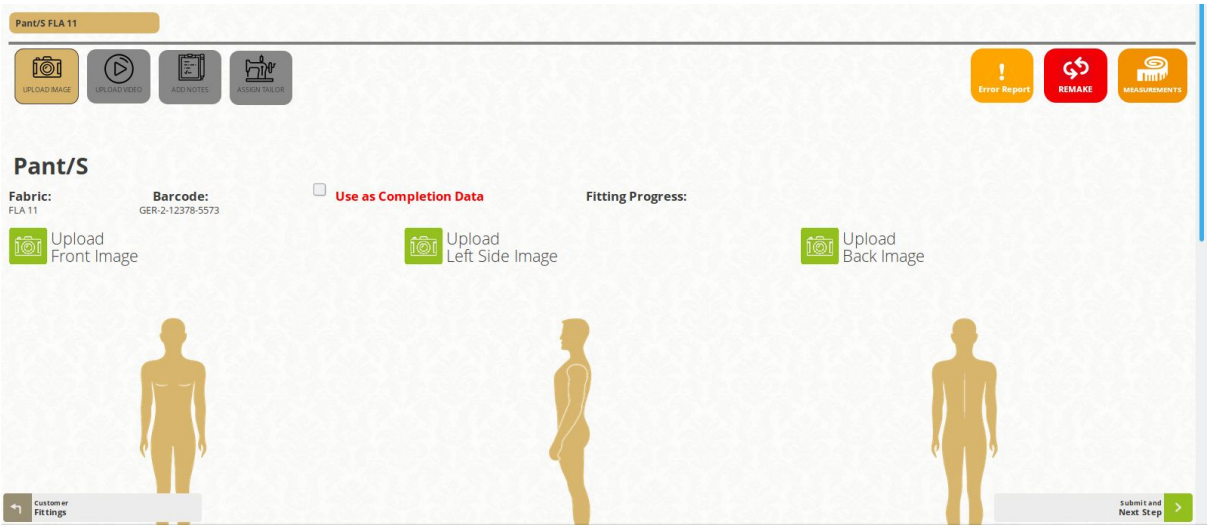
Nesta seção, é mostrada a etapa de seleção de medida corporal de um cliente durante pedido.

6.4. Tela de Pedido 3



Nesta seção, é mostrada a etapa de personalização de design de uma manga de um terno durante um pedido.

6.5. Tela de Fitting



Como explicado na seção 3.6, o fitting é um processo onde o cliente prova a peça de roupa e checa se ela está de acordo com o que ele solicitou (por exemplo, no tamanho certo e com o design escolhido etc). Caso haja algum defeito, é nesta tela onde o vendedor deve tirar uma foto com o cliente vestindo a roupa de diferentes pontos de vista (de frente, de lado e de costas)

7. Referências

[1]: Figura do Terno. Disponível em: <https://connor.imgix.net/Connor/Products/C17SJ308_BLU_1.png?w=480&h=680&bg=f1f1f1&cs=tinysrgb&fm=jpg&rect=440,0,2120,3000>. Acesso em: Acesso em: 9 de mar. 2019

[2]: Figura da calça. Disponível em: <https://d2h2vnfmmg5sct.cloudfront.net/catalog/product/large_image/09_408494.jpg>. Acesso em: Acesso em: 9 de mar. 2019

[3]: Figura do colete. Disponível em: <https://4.imimg.com/data4/UU/RK/MY-26774607/men-s-vest-coat-500x500.jpg>. Acesso em: Acesso em: 9 de mar. 2019

[4]: Figura da camisa. Disponível em: <https://5.imimg.com/data5/IM/GX/MY-10973479/mens-plain-formal-shirt-500x500.jpg>. Acesso em: Acesso em: 9 de mar. 2019